

**Structural and Parametric Optimisation
of (non)linear Dynamical Systems
using Genetic Programming**

A tool for automated controller design

Master's thesis
by D.J.H. Bruijnen
DCT 2003.75, CTB 555-03-1379

Engineering Thesis Committee:

prof.dr.ir. M. Steinbuch (supervisor)
ir. I.A.C. Soute (coach at Philips CFT)
dr.ir. M.G.J. van de Molengraft (coach at TU/e)
dr.ir. G.Z. Angelis (Philips CFT)
dr.ir. D.A. van Beek (TU/e)

Eindhoven University of Technology (TU/e)
Department of Mechanical Engineering
Control Systems Technology Group

Philips CFT
Mechatronic Equipment
Motion Control Group

Eindhoven, August 2003

Preface

This research was a project proposed by and carried out at the Motion Control Group of Philips CFT. It is a first exploration whether Genetic Programming is suitable for system identification and controller design. Preceding research has been carried out by Iris Soute who has used Genetic Programming for finding Lyapunov functions to prove stability.

My coaches have been Iris Soute (Philips CFT) and René van de Molengraft (TU/e). Many thanks to them for their good advise and support.

My project can be divided into three main elements: (1) Genetic Programming/Algorithms, (2) (OO-) Programming, (3) system identification/controller design. I would like to thank the next people from Philips CFT for their support to the project: Daniël Vangheluwe (1), Jos Onokiewicz (2), Georgo Angelis (1,3), Felix Peeters (3) and Emiel Botermans (3).

Furthermore, I would like to thank some other students doing also their traineeships/graduation project here for having interesting discussions about their projects, my project and other stuff; My neighbor Rick Scholte who knocks on the wall sometimes and my roommate Angelique Kessels who taught me the art of juggling during the rest breaks of WorkPace.

Contents

1	Introduction	6
2	Genetic Programming	7
2.1	Introduction	7
2.2	Representation	7
2.2.1	Structure	7
2.2.2	Nodes	8
2.3	GP algorithm	8
2.3.1	Initialization	8
2.3.2	Evaluation	8
2.3.3	Stop criterion	8
2.3.4	Selection	9
2.3.5	Genetic operations	9
3	GP research	10
3.1	System identification with GP	11
3.2	Controller design with GP	11
3.3	Other applications	11
4	Implementation of GP	12
4.1	EO-library	12
4.2	Matlab/Simulink	12
4.3	Features of the GP program	13
4.4	Conversion of a node to a Simulink-block	13
4.5	Discussion	14
5	Multidimensional optimization of a nonlinear function	15
5.1	A need for numerical optimization	15
5.2	Deterministic methods	16
5.3	Stochastic methods	17
5.4	Choice of method	17
6	GP & system identification	18
6.1	Introduction	18
6.2	Fitness-function	19
6.3	Results	19
6.3.1	System identification of a mass-spring-damper system	19
6.3.2	System identification of a fourth order system	20
6.4	Summary	22
7	GP & controller design	22
7.1	Introduction	22
7.2	Fitness-function	22
7.3	Results	25
7.3.1	Controller design for a mass-damper system	25
7.3.2	Controller design for a fourth order system	26
7.4	Summary	30
8	Conclusion and recommendations	31

List of figures	32
List of symbols	33
Abbreviations	33
Bibliography	34
A S-functions	35
A.1 calcstop	35
A.2 protecteddiv	36
B Fitness-function m-file	37
C GA implementation	46
D GP program	47
D.1 User manual	47
D.1.1 Gp4cd-files	47
D.1.2 Description of files	48
D.1.3 Node types	52
D.1.4 Performing a GP run	54
D.1.5 Troubleshooting	55
D.2 C++ code description	55
D.2.1 C++-files	55
D.2.2 Description	56
D.2.3 Known bugs/possible improvements/experience	57

Abstract

Ever since digital controllers became both available and cheap, a complex controller can be implemented with only a little bit of more effort. However, the design of such a controller is more difficult. Often a mere PID-controller is used, because it is easy, well understood and the performance is known.

In this thesis a controller synthesis method has been developed which can be applied to a wide range of plants. It is based on Genetic Programming (GP) which creates controllers with variable structures. Nonlinear dynamics can be included in the controller and/or the plant. Requirements for the controller are specified in time and/or frequency domain. The requirements are translated in a fitness-function, which is problem-dependant. This fitness-function returns a number indicating how good a certain controller meets the requirements. This fitness-function will be minimized by the evolution process of GP, and thereby finding the optimal solution for the problem.

GP is investigated in detail with respect to how it works, what kind of problems can be tackled with it and how to implement it. For this, a computer program has been written which is actually a structural and parametric optimization algorithm for dynamical systems using a GP/GA-hybrid (GA = Genetic Algorithm). First, some system identification cases are studied to get familiar with the GP program. After that, the GP program is transformed to perform controller design which is the main objective of this thesis. Some simple problems have been studied to test and optimize the GP program.

Some promising results are achieved but there is still a lot of research left to be done, e.g. with respect to nonlinearities, discretisation and MIMO systems. This thesis is a first exploration whether GP is suitable to create controllers for a user-provided plant. A futuristic/idealistic goal is to have an application which can generate an optimal controller for an arbitrary plant. However, in achieving this goal THE main challenge is to define our control objective into a GP fitness-function. After that, let evolution do the job. Pressing the start button would be the only skill needed...

Samenvatting

Na de opkomst van digitale regelaars zijn deze drastisch in prijs gedaald. Een complexe regelaar structuur kan hierop eenvoudig geïmplementeerd worden. Echter het ontwerpen van een dergelijke regelaar kost meer inspanning. Vaak wordt er voor een eenvoudige PID-regelaar gekozen, omdat deze eenvoudig en begrijpelijk is met een goede prestatie als resultaat.

Tijdens dit afstudeeronderzoek is een programma ontworpen waarmee regelaars gegenereerd kunnen worden voor een grote diversiteit aan systemen. Het is gebaseerd op Genetisch Programmeren (GP) dat regelaars creëert met een variabele structuur. Niet-lineaire dynamische elementen kunnen toegevoegd worden aan de regelaar en/of het te regelen systeem. Verder is het mogelijk om eisen te specificeren in het tijd- en frequentie-domein. De eisen worden vertaald in een fitness-functie die probleem-specifiek is. Deze fitness-functie retourneert een waarde die aangeeft hoe goed aan de gestelde eisen voldaan wordt. Deze fitness-functie wordt vervolgens geoptimaliseerd door GP wat dus het probleem oplost.

GP is uitvoerig onderzocht betreffende de werking, de problemen die ermee aangepakt zijn door andere onderzoekers en hoe GP geïmplementeerd kan worden. Vervolgens is er een computer programma geschreven wat in feite een structuur en numerieke optimalisatie algoritme is voor dynamische systemen met als achterliggend algoritme een combinatie van GP en GA (GA = Genetisch Algoritme). Ten eerste is er systeem identificatie mee uitgevoerd om vertrouwd te raken met het programma. Daarna zijn enkele wijzigingen aangebracht zodat het geschikt is voor het ontwerpen van regelaars. Dit is het hoofddoel van dit afstudeeronderzoek. Enkele eenvoudige problemen zijn voorgelegd aan het GP programma om het te testen en aan te passen waar nodig is.

Goede resultaten zijn behaald, maar er is nog veel onderzoek wat nog uitgevoerd moet worden. Denk hierbij b.v. aan niet-lineariteiten, discretisatie en MIMO systemen. Dit afstudeeronderzoek is een eerste aanzet om te onderzoeken of GP geschikt is om regelaars te genereren voor een willekeurig systeem. Een futuristisch/idealistisch doel is om een programma te hebben dat een optimale regelaar creëert voor een willekeurig systeem. Alvorens dit doel te bereiken, is de grootste uitdaging het definiëren van de regelaar eisen in een GP fitness functie. Daarna zal de evolutie van GP de rest doen. De enige benodigde vaardigheid zou dan op de startknop drukken zijn...

Structural and Parametric Optimisation of (non)linear Dynamical Systems using Genetic Programming

Dennis Bruijnen

Eindhoven University of Technology (TU/e)
Department of Mechanical Engineering
Control Systems Technology Group

Philips CFT
Mechatronic Equipment
Motion Control Group

1 Introduction

The design of a controller depends on the dynamics of the plant and the desired requirements. They are designed by control engineers who mostly implement a PID-like SISO controller. It is a simple and understandable control scheme, easily tuned and implemented. However, in case of highly coupled and non-linear system behavior this SISO PID approach is not always possible.

Since digital controllers are available and cheap, a more complex controller can be implemented with only a little bit of more effort. However, the development of such a controller is more difficult. In this thesis a method is developed to let an evolutionary computer program generate such an atypical controller. The user only provides the plant (in this case a Simulink model) and the desired requirements and the computer program will generate a suitable controller.

The mechanism used in this computer program is called Genetic Programming (GP). It is an algorithm that optimizes a variable structure according to a user-specified fitness-function. From an engineering point of view, this fitness-function is the most challenging part of GP. The fitness-function and the rest of GP are explained in Chapter 2.

The first objective of the graduation project is to investigate how GP works. The second objective is to implement it and the last objective is to assess the GP potential by performing system identification and controller design. The system

identification examples serve as a test case for GP, whereas controller design is our main objective. G. Naus [8] has continued the research of our GP-based system identification.

As this is our first exploration of GP, our focus is on linear systems. Therefore, it will be easy to judge GP results in view of the well-known results of linear system identification and control.

Information about GP is gathered to examine its working principle (see Chapter 2 and 3). Next, a generic GP computer program has been written (see Chapter 4), which optimizes dynamical systems. The first tests with system identification pointed out that GP had difficulties with optimizing the parameters of a model. To improve this, a numerical optimization method was added to the GP program (see Chapter 5). In Chapter 6, results of performing system identification with GP are shown. Finally, controller design has been carried out which is described in Chapter 7).

2 Genetic Programming

2.1 Introduction

"Genetic Programming" (GP) [20] is a part of a very large body of research called "Machine Learning" (ML). This term was first introduced by Samuel [Samuel, 1963] who meant "computers programming themselves" with it. A good contemporary definition of ML is: The study of computer algorithms that improve automatically through experience [Mitchell, 1996]. GP aspires to do precisely that; inducing a population of computer programs that improve automatically as they experience the data on which they are trained.

GP was introduced by Koza in 1992 [17]. The underlying basis is "Genetic Algorithms" (GA) [4]. The major difference between GA and GP is the representation of the solution. GA returns a fixed-length sequence of binary numbers which has to be interpreted. GP returns a ready-to-run program with a variable length. This is the power of GP; it is able to adapt itself to the complexity of the given problem. GP and GA are about telling the computer what needs to be done and not how to do it.

GA and GP are inspired by natural evolution. The body of research of algorithms inspired by natural evolution is called "Evolutionary Algorithms" (EA) [9] (which is a part of "Machine Learning"). The idea is that a population evolves using genetic operations like crossover, mutation and reproduction. The selection of individuals to produce offspring depends on their fitness. The fitness is a number which indicates how good an individual satisfies a problem.

Roughly, the fittest individuals survive and new, possibly better individuals will be created. Fitness-weighted selection and genetic operations make it possible to create offspring with a better fitness. This mechanism can be continued until the performance of the fittest individual is satisfying. Just like in nature it is survival of the fittest.

2.2 Representation

GP produces computer programs. To be able to solve problems, the computer program needs one or more inputs and an output. The con-

nection between the inputs and output must be syntactically correct. Furthermore, small parts of that structure have to be exchangeable so genetic operations can be applied.

2.2.1 Structure

Koza [17] proposed to use a tree structure consisting of functions and terminals. This way arbitrary programs can be built. Modifying a program is easily done by substituting branches or replacing nodes. Maintaining the tree structure will prevent syntax errors. An example of a tree structure is shown in Figure 2.1.

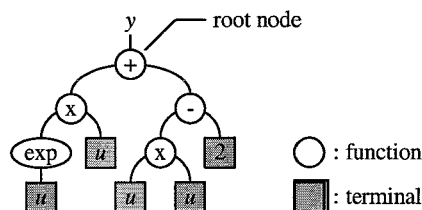


Figure 2.1: An example of a tree structure, $y = ue^u + u^2 - 2$.

A tree consists of nodes connected with each other. The root node of the tree returns the output of the computer program. A node can have an arbitrary number of inputs. If a node does not have any inputs, a so-called terminal, it will stop that branch from growing. Starting with the root node, randomly selected nodes can be attached until all branches have stopped growing because of terminal nodes. The tree depth is defined as the number of layers in a tree, e.g. the tree in Figure 2.1 has a tree depth of 4. Now a genetic operation can be applied. These are described in Section 2.3.5. More detailed information about genetic operations can be found in [20].

A tree structure is not the only way to represent a function. In literature a lot of different architectures have been proposed. Roughly, the main categories are tree structures, linear structures and graph / network structures (see Section 5.2 of [20]).

2.2.2 Nodes

There are two types of nodes: functions and terminals. Examples of functions are boolean functions (*and, or, not*), arithmetic functions (+, -, x, /), transfer functions ($\frac{1}{s}$, $\frac{1}{s+1}$) and many more. Examples of terminals are user-provided inputs and constants. The function set and the terminal set can be chosen freely provided that no syntax errors occur when nodes are connected arbitrarily.

2.3 GP algorithm

The GP algorithm starts with creating the first generation of a new population containing a user-defined amount of randomly generated programs (also called individuals). The fitness of these individuals is evaluated. The GP algorithm stops if a termination criterion is satisfied. If not, parents are selected in a probabilistic way according to their fitness. A new generation is created by applying genetic operations. The fitness of the individuals in the new generation is evaluated. The loop is now closed and is continued until the termination criterion has been reached. An overview of this GP algorithm is shown in Figure 2.2.

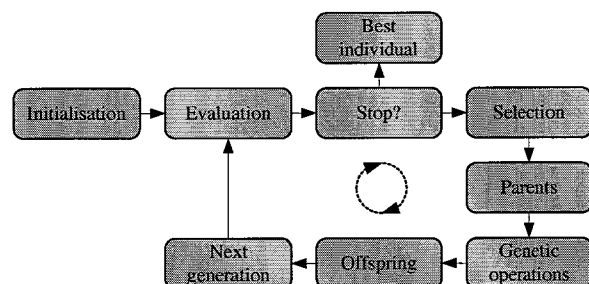


Figure 2.2: An overview of the GP algorithm.

To set up GP for a particular problem, the main five items which have to be defined are:

- Function set
- Terminal set
- Fitness-function
- Evolutionary parameters
- Termination criterion

When these items have been defined well and a solution for the particular problem exists, then there is a realistic chance that the GP algorithm finds a good solution. Mainly, it is a matter of time and resources. Because GP is a stochastic search technique to find an optimum solution in a wide search space it does not always find the desired solution after one run of the GP program. GP produces different solution every run because of its stochastic nature, so it is possible that a better solution is found in another run.

2.3.1 Initialization

The GP algorithm starts with creating a new population. The individuals are created randomly using nodes of the function set and the terminal set. The amount of individuals in a population is a tunable parameter. Using more individuals gives a more diverse population but is also more computational demanding. The maximum allowed complexity can be set by limiting the tree depth. Increasing the maximum tree depth increases the search space. The search space should cover the complexity of the given problem, but increasing it too much can cause convergence problems.

2.3.2 Evaluation

The performance of an individual has to be determined. This is done by defining a fitness-function. The fitness of all new individuals in a generation has to be evaluated. This part is by far the most computational demanding compared to the rest of GP algorithm. The fitness-function can be chosen freely. A high-level specification of the requirements can be implemented here. A well-defined fitness-function is crucial in obtaining a good result with GP. It shapes the landscape of the search space.

2.3.3 Stop criterion

A stop criterion is necessary, or else the algorithm will continue searching for better results forever. Two commonly used criteria are: "stop if a user-defined number of generations has been reached" or "stop if the best-of-run fitness meets a user-defined value". Manual termination of the program is also possible.

2.3.4 Selection

If the stop criterion is not met, a new generation has to be created. Individuals (parents) are selected to produce offspring. An often used selection method is selecting parents probabilistically in proportion to their fitness. Another popular selection method is "Tournament Selection". This is simply selecting the best individual out of a few randomly selected individuals from the population. This is repeated until enough parents are selected.

Two breeding policies are often used: Generating a whole new generation using genetic operations is called "Generational GP". Replacing only a few individuals by newly generated individuals is called "Steady State GP". Other breeding policies can be found in [21].

2.3.5 Genetic operations

Genetic operations are used to create new individuals (offspring) out of existing individuals (parents). These operations are based on mechanisms seen in nature during evolution. Commonly used genetic operations are crossover, mutation and reproduction.

Crossover

The GP structures of two selected parents are taken. A randomly selected branch of both parents will be exchanged. Now two new individuals have been created by using parts of both parents (Figure 2.3).

The usage rate of genetic operations can be set for each type. Crossover is the most often used genetic operation in practice. A usage of about 70% is common. The impact on the individual's fitness is huge. Big steps are made through the search space. Although crossover has a negative effect on the fitness in over 75% of the cases, it is useful in making leaps in the search space, thereby avoiding ending in a local optimum.

Mutation

Two mutation operations are "point mutation" and "branch mutation". Point mutation selects one randomly chosen node from the structure of a parent and replaces it by a new node (Figure 2.4). This node should have the same

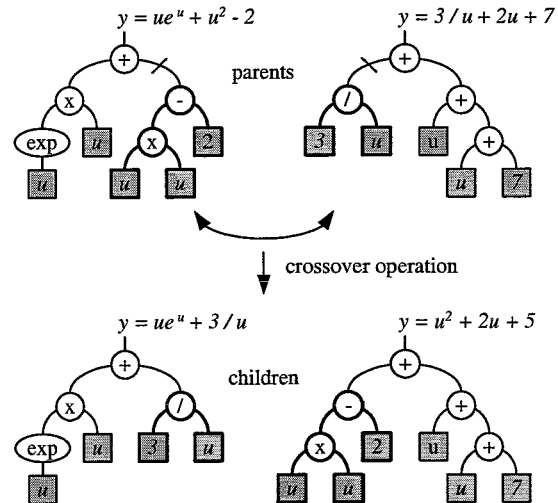


Figure 2.3: An illustration of a crossover operation.

amount of inputs to avoid connectivity errors. Branch mutation selects one randomly chosen node and creates a new branch at that place.

These are only two examples of commonly used mutation operations. Much more types of mutation operations exist. The main condition is that they should replace some part of a structure with new, randomly selected nodes. A mutation operation can change the size of the tree (but not necessarily) and furthermore, it prevents extinction of certain node types.

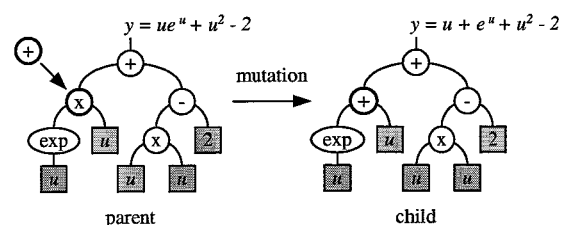


Figure 2.4: An illustration of a mutation operation.

Mutation is the second most used genetic operation with a common usage of 30%. It has a less destructive effect than crossover, because on average it makes smaller steps through the search space. It also keeps the diversity of node types in a population high, because it randomly inserts new nodes from the terminal and function sets.

Reproduction

Reproduction is simply creating a new individual by making an exact copy of the selected parent (Figure 2.5).

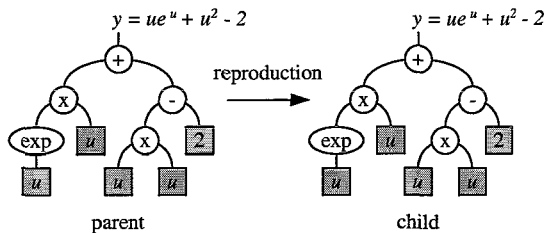


Figure 2.5: An illustration of a reproduction operation.

Reproduction is useful to preserve the best individual, also called "Elitism". Research has pointed out that the reproduction operation does not add a lot of performance gain to GP, because it does not change the behavior of an individual. a low usage of 2% is common.

Other genetic operations

Other genetic operations which have been used during this research are: expansion mutation, numeric mutation and branch addition. Expansion mutation replaces a terminal node by a randomly created branch. Numeric mutation changes all parameters of an individual randomly. Branch addition creates a new tree and combines that with the old tree by joining them with a 2-input node. The first two have been removed at a later stage, because they appeared to be obsolete. Branch addition is still present, because it stimulates linear combinations of parts which is often beneficial in the field of linear controller design. Simulations have shown that it has a positive effect on the performance of GP.

More information about genetic operations can be found in [20]. In the next chapter recent research is presented in the field of GP.

3 GP research

GP has existed only for about a decade. Since the foundation was laid in the early 1990's the popularity of GP has been increasing enormously because of its promising results. Also the fast speed increase of computing power is an important reason, because GP is very computational intensive. A lot of books, articles and papers have been dedicated to the subject. Because of the flexibility of GP to tackle an arbitrary problem, it has been applied to a wide range of problems. Already a lot of results have been achieved which are equal or better than results produced by experts in the field.

References that have been used here to gather information about GP are [15], [20] and [21]. Also a lot of information can be found on the internet: e.g. [11], [12] and [13].

GP research can be broadly categorized into two groups: 1. GP Techniques and Theory, 2. GP Applications. An overview is given from the latter only (see Figure 3.1), because in this research, GP is used as a tool for system identification and controller design. For both topics an example is given in the next two sections. They are compared with the approach in this thesis.

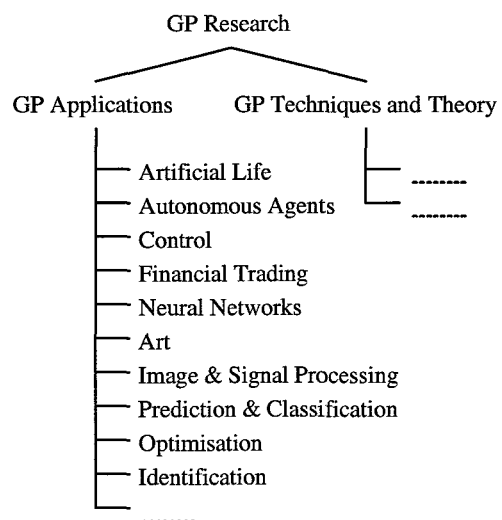


Figure 3.1: Overview of GP research.

3.1 System identification with GP

"Structural system identification using genetic programming and a block diagram oriented simulation tool" [6]

In this paper, nonlinear system identification using GP is investigated. Models are created in Matlab/Simulink using continuous-time blocks (s-domain). A plant is created containing a second order transfer function, a saturation and a time delay. Time-domain input-output data is created by simulating the model. The chosen input involved steps and ramps. The same input data is given to generated models by GP. The fitness-function is defined as the sum of the squared error between the output of the plant and the GP model. A combination of simulated annealing and nelder-simplex optimization was used to support GP with the parameter optimization.

This research is very similar to the system identification approach in this thesis; Matlab/Simulink is used for simulations and comparison of models in the time-domain. The major difference is a different parameter optimization routine. Further, only linear system identification has been carried in this thesis, because our main objective was to perform controller synthesis. Although, it would not be such a big step to transform the GP program to a nonlinear system identifier. It is just adding some nonlinear blocks in the function set and choosing the input cases well to ensure observability.

3.2 Controller design with GP

"Automatic Creation of both the Topology and Parameters for a Robust Controller by Means of Genetic Programming" [16]

In this paper, a robust controller has been developed with GP for a second order system with no zeros. Controllers are generated and examined using the SPICE simulator (tool to solve differential equations). The fitness-function consists of 10 components. 8 components are based on an ITAE (Integral of Time-weighted Absolute Error). For these 8 components, 4 different parameter sets and 2 different trajectories

are used to obtain a robust controller. The 9th component is the examination of the closed loop (determined using an AC sweep). A gain bigger than 3 dB in the low-frequency area is penalized. The 10th component is a stability test. A small pulse is applied at the start. A penalty is added to the fitness-function when the output exceeds the pulse's amplitude at the end of the simulation. This is a stability test.

The major differences with this thesis are the fitness-function and the simulation environment. The same authors of the paper have produced some more papers about controller design with GP. The applications are different, but the fitness functions is about the same.

3.3 Other applications

GP has also been applied to a lot of other applications. A few examples about totally different applications are given next:

"Using Genetic Programming to find Lyapunov functions" [15]

This paper is about finding Lyapunov functions which are used to prove stability for (non)linear systems.

"Better Trained Ants for Genetic Programming" [21]

In this paper, an artificial ant which has to follow the "Santa Fe trail" is programmed. The "Santa Fe trail" is a virtual path on a check board with food at specific locations. The ant should follow the optimal path and eat all the food.

"Genetic Programming Applied to Traffic Control" [14]

In this paper, a set of crossing roads with traffic lights at the intersections have to be controlled. The problem of controlling the traffic lights optimal in terms of traffic rate and waiting time of a car at an intersection is presented to GP.

The working of GP has been studied and the fields where GP has been applied is investigated. In the next chapter, the implementation of the generalized GP program is discussed. This application can be used for both system identification and controller design.

4 Implementation of GP

4.1 EO-library

It is possible to build the whole GP program from scratch. This would result in a lot of work which has already been done by others. There are some libraries available on the internet which are dedicated to Evolutionary Algorithms. A few examples are GPC++ created by A. Fraser (1994) [1] at the University of Salford and EO (Evolving Objects) by the Geneura Team at University of Granada (1998) [10].

In this thesis the EO-library is chosen, because it appears to be the most complete and flexible library of the considered libraries. EO is a template-based, ANSI-C++ compliant evolutionary computation library (description of EO: [18], reference manual of EO: [19]). It was developed to implement evolutionary algorithms like Neural Networks (NN), Evolutionary Strategies (ES), Simulated Annealing (SA), Genetic Algorithms (GA) and Genetic Programming (GP).

4.2 Matlab/Simulink

System identification and controller design will be performed with GP in this thesis. A good choice has to be made for the representation of a solution. It is preferred to create models in a user familiar form so the result can be easily interpreted and used for further application. Also the evaluation of models should be easy and robust. That is why it is decided to use Simulink-models. The so-called Matlab Engine is shipped with Matlab. This enables a C++ application to communicate with Matlab/Simulink [3], [7]. This approach has a number of advantages:

- The GP algorithm can be implemented in C++ instead of implementing everything in Matlab. This will improve speed. Also the available EO-library can be used to implement GP which saves programming time.
- All available tools of Matlab and Simulink can be used to evaluate models and evaluate fitness cases. Implementing algorithms in C++, for example to solve differential equations, is not needed any more. Matlab/Simulink has a lot of sophisticated tools

for that, which are optimized during the years.

- The result is a ready-to-run Simulink model which can be easily interpreted and can be used for further application.
- Most controller engineers are familiar with the Matlab/Simulink environment.

When combining C++, the EO-library and the Matlab Engine for access to Matlab and Simulink, all ingredients are available to create a GP program which optimizes dynamical systems according to a user-defined fitness-function. This fitness-function has to be defined in Matlab so the GP program will perform a specific task, like system identification and controller design. The fitness-function alone decides what task is carried out, the GP program only optimizes dynamical systems according to this fitness-function.

A global view of the architecture of the implemented GP algorithm is shown in Figure 4.1. The source code can be found on a cdrom (see [2]), an example of a fitness-function m-file can be found in Appendix B.

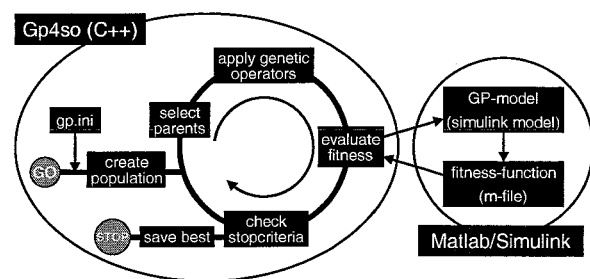


Figure 4.1: A global view of the architecture of the implemented GP algorithm.

The implementation of the GP program and how it should be used, is described in detail in Appendix D. From now on one run of the GP program is called a "GP run".

4.3 Features of the GP program

A brief list of important features of the GP program is listed next. More information about the GP program can be found in Appendix D.

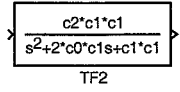
- The representation used is a tree structure for both system identification (Chapter 6) and controller design (Chapter 7).
- The fitness-function returns a value related to the performance of an individual. To achieve this, an individual in the GP program has to be converted to a Simulink file which is then evaluated in Matlab/Simulink resulting in a fitness value.
- Available genetic operations are: crossover, branch mutation, point mutation and branch addition.
- A Genetic Algorithm has been implemented to boost the numerical optimization. It is implemented as a genetic operation of GP. The effect and motivation of this GA is discussed in Chapter 5.
- Parents are selected by Tournament Selection. Both "Steady State GP" and "Generational GP" are possible.
- The stop criterion is threefold. The maximum number of generations can be set. A minimum fitness level can be set and the GP run can be aborted manually.
- The GP program has got several tuning parameters. These can be changed in a so-called ini-file. (see Appendix D)
- The GP program is divided into three sections: initialization, performing the GP run and saving useful data such as the fitness progression, the GP settings and the best model.
- A GP run can take days to evolve. If a fitness improvement occurs, then that model together with its fitness is saved to a file. One reason is that the current best result can be shown during a GP run. Another reason is that if the application crashes the most useful information is kept.

- Sometimes it is not possible to run a randomly created Simulink-model. The main reason is numeric problems, like singularities and stiff or even unstable models. This leads to slow progression of the Simulink-simulation or even abortion of it. To catch these faults an S-function has been written which stops the simulation after a specified number of seconds (code is presented in Appendix A). Further, also a check is made after each simulation for errors using the Matlab-command "lasterr" which returns the last occurred error. If the simulation was not successful a high fitness is assigned meaning a "bad" individual.
- The chances of selecting genetic operations are independent of each other. So it is possible that none or more than one genetic operations are applied to the same model of a single generation.

4.4 Conversion of a node to a Simulink-block

GP builds a tree from nodes. Each type of node has different properties, like the amount of inputs and the amount of parameters it contains. If GP creates a tree using these nodes, this has to be converted to a Simulink model to be able to evaluate it. Each node is converted separately and connected as indicated by the tree. In Appendix D.1.3, the Simulink blocks are presented belonging to all available node types, which are: Add, Subtract, Multiply, Divide, Integrate, Differentiate, TF 1st order, TF 2nd order, Gain, Constant, Time Delay, Input0, Input1 ... Input7. This can easily be expanded. All available Simulink blocks in the Simulink Library could be used if they would be needed. Also, Simulink makes it possible to define your own block by creating a so-called S-function.

An example of a second order transfer function is shown in Figure 4.2. Explanation of the chosen structure can be found in Appendix D.1.3. For each node type, the Simulink block and the formula is given. Parameters of a block are initialized with different probability distributions to create more suitable blocks. n is the user-specified maximum initialization value.

TF 2nd order:

$$y(s) = c_2 \frac{c_1^2}{s^2 + 2c_0c_1s + c_1^2} u(s)$$

c_0 : uniform $[0, 1]$
 c_1 : uniform $[0, n]$
 c_2 : loguniform $\pm[\frac{1}{n}, n]$

Figure 4.2: Properties of a second order transfer function.

4.5 Discussion

Conditioning and normalization

The reason of designing blocks in a special form is to create useful models more frequently. If this was not done, initially, too much initialized models would result in stiff or even unstable models. It is obvious that this has a negative effect on the performance of GP. During the first tests of system identification only a single uniform probability distribution between $-n$ and n was used to initialize the parameters. Furthermore, the parameters were not positioned in the transfer function to take account of normalization and the difference in order of magnitude between all coefficients of a transfer function. For example, the parameters of a second order transfer function like $\frac{c_3}{s^2 + c_1s + c_2}$ are often set to obtain an under-critical-damped transfer function. In that case, the order of magnitude of c_2 is about the order of magnitude of c_1 squared.

In Appendix D.1.3 the special form of each block is shown. The tests after these adjustments have shown the expected improvements. Actually, designing the blocks in such a way is equal to inserting knowledge about the problem and lighten the task of GP a little bit.

The parameters have been put in such a way that they are conditioned well. Most of them have the order n and a few the order 1. Furthermore, all transfer functions have been normalized to a static gain of 1 with an extra static gain parameter multiplied with the transfer function. This parameter is initialized with a loguniform

probability distribution so the chance of amplifying or reducing the gain is equal.

This could also be solved by GP, as GP can add a gain-block in front of a transfer function block. The reason that a transfer function is given a static gain parameter is to give the transfer function more freedom so it can have any shape for that type of transfer function. A disadvantage is that it adds a parameter extra in one node, so the pressure is put a little bit more at the numerical optimization of the parameters (see Chapter 5) instead of GP. In this case it does not make a big difference because there are already more than one parameters in a transfer function node.

s-domain vs. z-domain

The main purpose of this research was the exploration of the suitability of GP for controller design considering easy plants. Results from conventional methods are known, so the performance of GP can be compared with these benchmark problems. For interpretability, the s-domain has been chosen to create models. In the future it could be of interest to switch to the z-domain. This is explained later on.

Having more parameters per node is not promotable when using only the GP algorithm for parametric and structural optimization, because GP lacks efficiency; GP only mutates whole nodes, so the chance it returns the same block with slightly different parameters is very small. However, having more parameters per node is unsurmountable for controller design in the s-domain using Simulink and a tree representation of an individual. The advantage of using the s-domain is that such a model is directly interpretable for control engineers. Disadvantages are, that discretisation is not accounted for and, that a transfer function can not be made with elementary blocks when using a tree-structure. A feedback coupling is needed to create a transfer function using elementary blocks.

In the z-domain elementary blocks can be used, which is preferable for GP. A network-structure could be used to generate models. No conversion from continuous to discrete is needed

when performing a simulation. Another issue is that now-a-days, most controllers are implemented in a discrete form, for example on a DSP. So using the z-domain already incorporates discretisation.

In short, the z-domain provides some nice advantages, but the s-domain has been chosen because of the interpretability. Due to this choice and the choice of using a tree structure representation for an individual, transfer functions have to be included to the function set and a parameter optimization algorithm is indispensable. Which additional parameter optimization routine to use is discussed in the next chapter.

5 Multidimensional optimization of a nonlinear function

5.1 A need for numerical optimization

During the first tests with the GP program some problems occurred. GP has been implemented using a tree representation for an individual (see Chapter 4). The problem is that transfer functions can not be made in a tree-structure using elementary blocks. The solution is to use first order and second order transfer function blocks, so poles and zeros can be placed arbitrarily by the GP program. However, GP is not able to optimize the parameters of those blocks with the genetic operations. GP only mutates whole nodes/blocks. A block like a second order transfer function can contain up to 5 parameters which is far from elementary. The chance that it mutates to the same block with slightly different parameters is very small so optimization of these blocks is almost impossible with the GP program.

For simple models it quickly finds the right structure, but it takes a long time before it finds the optimal parameters. GP then keeps on trying different models because it does not know that the right structure has already been found.

For more complex models, GP occasionally produces the right structure during the run, but the parameters are not yet optimized. As the chance is very small that GP also finds the optimal parameters, the model gets a bad fitness, and it will not survive in the next generation. The result is that GP probably stagnates in a local optimum. The diversity of the population decreases so the chance that the right structure with the correct parameters will be found becomes smaller.

A solution to this problem is to use a dedicated parameter optimization routine. Optimizing a dynamic model according to a calculated fitness-function is actually optimizing a multidimensional nonlinear function. In literature a lot of methods are presented. Mainly they can be divided into two parts: deterministic methods and stochastic methods. There is no best-of-all method. Which method is the best to use is problem-dependent.

5.2 Deterministic methods

Local gradient

Deterministic optimization methods can be divided into two types; methods which use only function-evaluations, for example the Nelder-Mead Simplex method, and methods which use also derivative information, for example the Conjugate Gradient method. These two examples are explained briefly in this section. More information can be found in [23].

1D optimization

All gradient methods are based on optimizing a function in one direction and then choose another direction and so on. Two commonly used methods are Parabolic Interpolation and Golden Section Search. The first one chooses three points and fits a parabola through it. The minimum of this parabola is added to the points and the outer point is thrown away. Now another parabola is fitted through these three points and so on. This method works good for sufficiently smooth functions.

Golden Section Search is designed to handle, in effect, the worst possible case of function optimization. It starts with finding a bracketing triplet. This means, finding three points with the middle point a lower fitness as the other two. Then another point is chosen at the golden section fraction ($= (3 - \sqrt{5})/2 \approx 0.38197$) between the outer points. This is the most optimal choice considering the worst case of function optimization. A derivation of this can be found in [23].

Both Golden Section Search and Parabolic Interpolation are only designed for local optimization near the starting point.

Conjugate Gradient method

A common error is to assume that any reasonable way of incorporating gradient information should be about as good as any other. This line of thought leads to the following not very good algorithm, the Steepest Descent method: Optimize in the direction of the gradient and do the same at the found optimum. For long, narrow valleys it will perform many small steps which is not very efficient.

A smarter way of choosing new directions is the concept of "non-interfering" directions, more conventionally called conjugate directions. A mathematical description can be found in [23]. The conjugate directions are chosen to try to ensure that the directions that have already been minimized stay minimized. Contrary to intuition, this does not mean following the line of steepest descent each time.

The conjugate directions are actually calculated on the assumption that the error surface is quadratic, which is not generally the case. However, it is a fair working assumption, and if the algorithm discovers that the current line search direction is not actually downhill, it simply calculates the line of steepest descent and restarts the search in that direction. Once a point close to a minimum is found, the quadratic assumption holds true and the minimum can be located very quickly.

Nelder-Mead Simplex method

The Nelder-Mead Simplex method is an optimization method that does not need derivatives. It does not make use of line optimization. The main concept used by the Nelder-Mead Simplex method is the geometrical concept of a simplex. A simplex is the geometrical figure consisting, in n dimensions, of $n + 1$ points (or vertices) and all their interconnecting line segments, polygonal faces etc. Examples are a triangle ($n = 2$) or a tetrahedron ($n = 3$).

What the Nelder-Mead Simplex algorithm basically does is to calculate the function at each of these vertices and order them according to the function value. The worst is discarded and a new vertice is proposed using rules of reflection, expansion and contraction. The other vertices are retained and only for the new vertice the function is evaluated. Then ordering takes place as above and the algorithm searches until no further improvement is gained.

The algorithm performs well for finding a local minimum. Also narrow valleys are no problem, the simplex first moves to the valley and then stretches itself in the direction of the narrow valley.

5.3 Stochastic methods

Simulated Annealing

Simulated Annealing is inspired by the cooling process of metals. When adjusting the cooling scheme of a metal in the right way, properties of the metal can be determined. As a metal cools, the atoms organize themselves into an ordered minimum energy structure. At a high temperature the atoms move a lot. At a lower temperature they move less which results in a fixed structure below some temperature level. If the temperature decreases slow enough, the atoms have time to arrange themselves in a fine structure (low energy). Decreasing the temperature very fast results in a coarse structure (high energy).

Simulated Annealing tries to imitate this process. It tries to find the global optimum of a multidimensional function by decreasing the "temperature" slow enough. When evaluating the fitness with a different state, this "temperature" determines the rate of accepting the state if it has a worse fitness. A better fitness is always accepted. At a high temperature, a worse fitness is often accepted which causes the state to travel from one optimum to another. Decreasing the temperature this will become less until it stagnates in an optimum. This is only a brief explanation of the method, in literature there is more information available [23].

How to alter the state is not defined by Simulated Annealing. Often a combination of the Nelder-Mead Simplex method with Simulated Annealing is used. It has shown to be a very effective method to optimize a multidimensional nonlinear function.

Genetic Algorithm

A Genetic Algorithm (GA) is highly related to GP. The main difference is the representation of the solution. GA only optimizes a fixed length representation of parameters. Because of this, it is much more efficient in optimizing parameters than GP. A lot of variants have been developed since the 60s, when GA was developed.

A description of such a variant of GA can be found in [4] and will be described briefly here. Like GP, GA also generates a population of in-

dividuals which is modified every generation using genetic operations. An individual is simply a collection of a fixed amount of numbers. Its fitness is determined by a fitness-function.

Each generation, the individuals with the best fitness are chosen to reproduce. These parents produce offspring using the crossover operation. Then the values of each offspring are mutated with a random number using a normal distribution of probability. For each offspring this standard deviation of the step size is increased or decreased by a factor 1.5. Parents pass on this standard deviation of the step size to its offspring. By selecting the best individuals, the standard deviation of the step size will adapt to the shape of the fitness landscape.

This method has shown to be very robust for a variety of problems.

5.4 Choice of method

Some consideration for the numerical optimization of the parameters of an unknown model are as follows:

- The amount of evaluations has to be reduced to a minimum, because each fitness evaluation contains one to several simulations in Simulink which costs time.
- Model parameters are randomly initialized so compared to the optimum values, a huge deviation is likely to be present initially. Because of this, a global optimization method is preferred instead of a local optimization method.
- The landscape of the fitness-function is unknown and probably highly nonlinear. Also the sensitivity of all parameters can vary a lot. The algorithm should be somehow self-adjustable to the right order of magnitude to change the parameters.
- The solution does not have to be the exact global minimum after one optimization. GP keeps on optimizing models by random selection of individuals from the population so it is possible that a model is optimized more than once.

The fitness landscapes can be unknown, non-linear, noisy or contain narrow valleys, so a robust optimization method is needed. Suitable candidates are the stochastic methods, Simulated Annealing and a Genetic Algorithm. Both are able to find global optima for a difficult fitness-function. A Genetic Algorithm has less problem-dependent settings than Simulated Annealing. Furthermore, a useful example of a genetic algorithm is already available and is easy to implement, so this algorithm is selected for optimizing the parameters of a model.

This Genetic Algorithm has been implemented in the GP program (see Appendix C). In the next chapter results of system identification are shown.

6 GP & system identification

6.1 Introduction

To examine the performance of our GP program (see Chapter 4), system identification is explored. Defining a fitness-function for system identification in the time-domain is relatively easy, so all effort can be put into the optimization of the computer program. A part of the field of system identification is trying to find the system dynamics by examining input-output data (black-box identification). From now on, the system to be identified is called the plant.

Terminal and function sets

The available nodes, which represent Simulink-blocks in a Simulink-model, can be divided into two groups: the terminal set: {constant, input} and the function set: {add, subtract, multiply, divide, integrate, differentiate, tf1, tf2, gain}. GP builds trees using these two sets. A terminal does not have any inputs so it finalizes a branch. Functions have one or more inputs so they cause a tree to grow. With these nodes GP will be able to identify linear systems. "tf1" and "tf2" are respectively a first order and a second order transfer function. Both blocks are needed to enable creation of both real/complex poles and zeros so every possible proper system can be build.

Result of a GP run

A GP run can be terminated several ways; a maximum number of generations, a certain fitness-level or manually. Then, the GP program will create a Simulink-file of the best GP model. This GP model can be validated by applying other inputs than the one used for determining the fitness. Also comparing the transfer function of the best model and the plant is very useful to assess the result.

6.2 Fitness-function

As explained in Chapter 2, a fitness has to be assigned to each GP individual to indicate its performance. Only time-domain evaluations are used during system identification, while exciting the GP model with a sine, a step or an impulse. Identification will be based on a set of m inputs. The output of the GP model should resemble the plant as much as possible for all m input cases. Therefore, a fitness-function proportional to the squared output difference between the GP model and the plant is used. A weighting factor w_j is applied to scale the order of magnitude of the fitness-contribution for input case j with respect to the other input cases. The total fitness-function is the sum of all input cases. In equation 6.1 the fitness-function is shown. y_{pj} is a vector with simulation data of the plant for input case j . This is a discrete time measurement with n samples. y_{mj} is the accompanying output of the GP model.

$$f = \sum_{j=1}^m w_j \sqrt{\frac{1}{n} \sum_{i=1}^n \{(y_{mj}(i) - y_{pj}(i))^2\}} \quad (6.1)$$

6.3 Results

Some simple systems have been taken to be identified by the GP program. An Intel Pentium IV 2,4 GHz with 256 MB RAM is used to perform the GP runs.

6.3.1 System identification of a mass-spring-damper system

System

The Simulink model shown in Figure 6.1 is used to create input-output data. It is a mass-spring-damper system with a damping coefficient of 0,2 and a natural frequency of 10 rad/s. In Figure 6.3 the three input cases are shown.

The measured output is contaminated with random noise to make the problem more realistic.

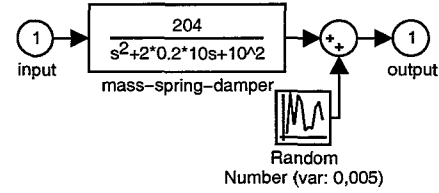


Figure 6.1: Plant to be identified.

Settings

The main settings of the GP program are shown in the table below.

data points	400
time span	3 s
inputs	step at 0,3 s 0,8 Hz sine 3,2 Hz sine
maximum tree depth	4 nodes
population size	100
function set	gain, add, subtract, integrate, 1st order TF 2nd order TF
terminal set	input
genetic operations	crossover (70%) branch mutation (10%) branch addition (5%) point mutation (15%) GA optimization (20%)
offspring	10 per generation using a tournament selection of 2

Results

The GP run was aborted manually, because the fitness did not improve anymore for awhile. Properties of the best individual are:

run time	40 minutes
best fitness	0,22
generations	92
fitness evaluations	$3,1 \cdot 10^4$
error	about equal to the inserted noise

The best model is shown in Figure 6.2. The structure is exactly correct. Only the numerical values vary a little bit.

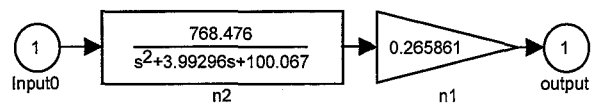


Figure 6.2: Best GP model.

In Figure 6.3 the plant and the best model are compared for all fitness cases. The error left

mainly originates from the inserted noise in the plant.

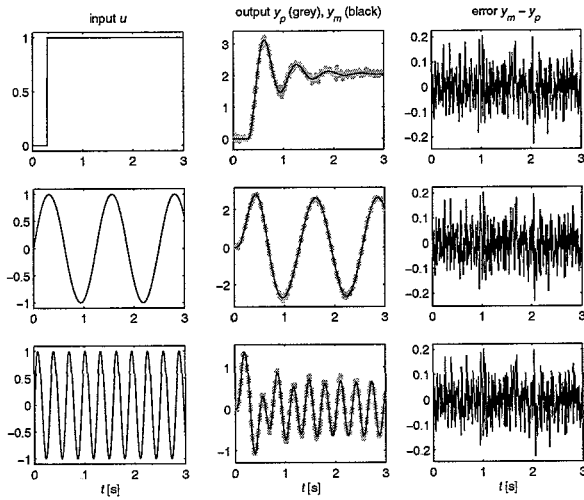


Figure 6.3: Comparison of the plant and the best model using the same input data as used in the fitness-function.

Validation

A good result is obtained when applying different data sets (see Figure 6.4). It can be concluded that the GP model is a good approximation of the plant.

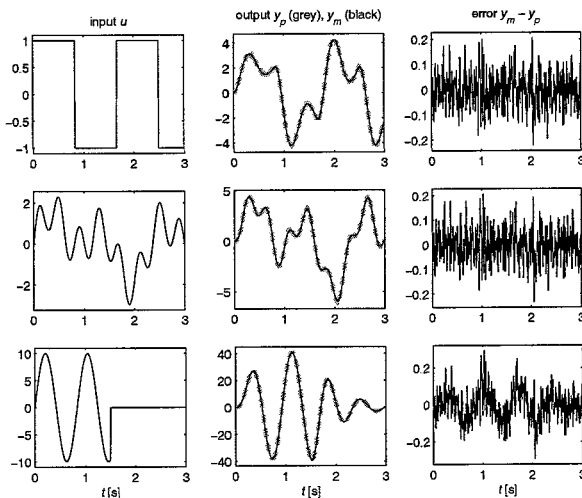


Figure 6.4: Validation of the best model using different input data.

6.3.2 System identification of a fourth order system

System

Two damped masses with a spring and a damper in between will be identified next (see Figure 6.5a). A Simulink model has been made which describes the relation between F and x_1 (see Figure 6.5b). In Figure 6.7 the 3 input cases are shown.

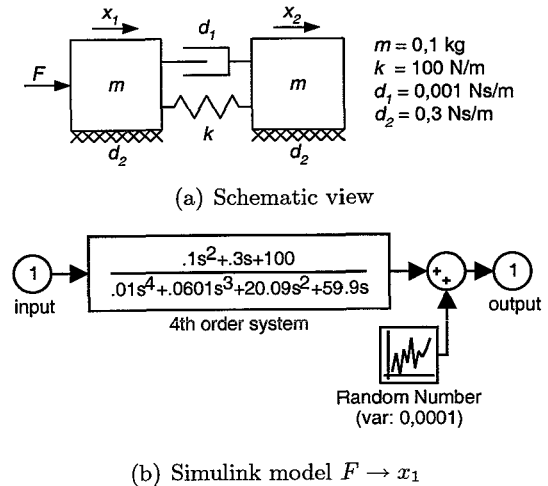


Figure 6.5: Plant to be identified.

Settings

data points	400
time span	3 s
inputs	step at 0,3 s 0,5 Hz sine impulse at 0,3 s
maximum tree depth	8 nodes
population size	100
function set	gain, add, subtract, integrate, 1st order TF 2nd order TF
terminal set	input
genetic operations	crossover (70%) branch mutation (10%) branch addition (5%) point mutation (15%) GA optimization (20%)
offspring	10 per generation using a tournament selection of 2

Results

run time	6,1 hours
best fitness	0,045
generations	319
error	impulse: high frequency component has not been caught

The GP run was aborted manually. The best GP model is shown in Figure 6.6. In Figure 6.7 the plant and the best model are compared for all fitness cases. The GP model performs good for these 3 fitness cases, but the structure is not the same. It mainly behaves like a second order system. The three second order transfer function do not exert a lot of influence on the result because of their low gains.

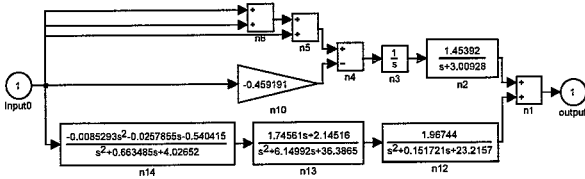


Figure 6.6: Best GP model.

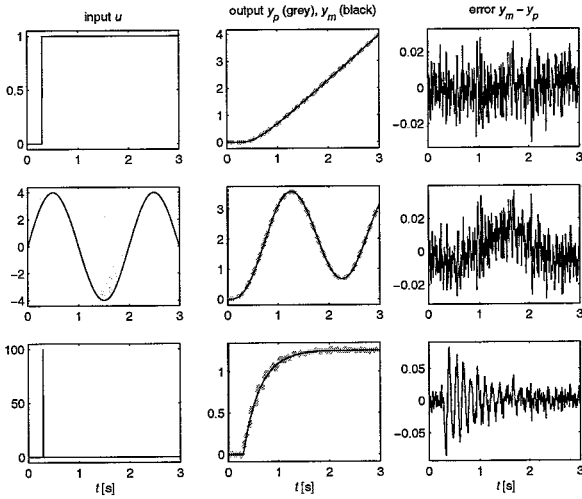


Figure 6.7: Comparison of the plant and the best model using the same input data as in the fitness-function.

Validation

The other input data is also applied to the GP model (see Figure 6.8). For this input data, the output of the GP model and the plant are also similar. Unfortunately the anti-resonance/resonance peaks were not caught. This is clearly visible when comparing the transfer function of the plant and the model (see Figure 6.9). The GP run has been performed sev-

eral times, but a good model structure including the anti-resonance/resonance peaks has not been found.

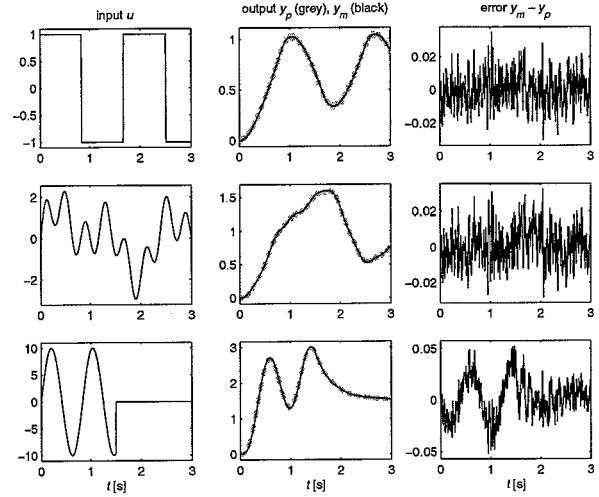


Figure 6.8: Validation of the best model using different input data.

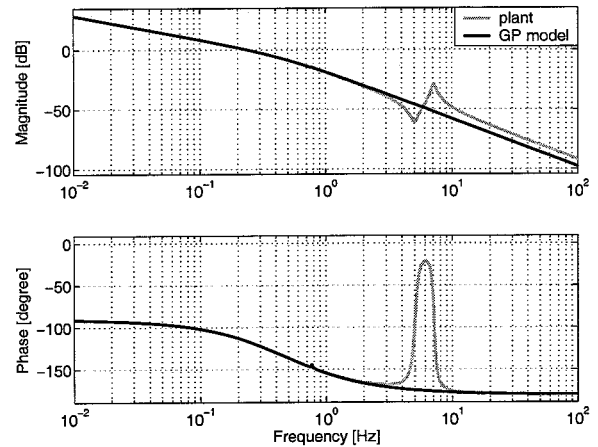


Figure 6.9: Transfer function (gain, phase) and coherence function

The reason that it does not find the correct structure in this case is observability. For all three fitness-cases the time-responses of the plant and the GP model coincide, so GP is not able to find the correct structure in the first place. To overcome this problem the fitness-function has to be defined better. A possibility for linear models is comparing the transfer-functions of the plant and the GP model.

6.4 Summary

GP is able to perform system identification for simple systems. The structure of second order systems presented in Section 6.3.1 was exactly reproduced, though small numerical variations were present. The fourth order system could merely be approximated. The exact structure was never found. The best GP model mainly behaved like a second order system which did not catch the anti-resonance/resonance peaks. The cause of this is that the dynamics of the anti-resonance/resonance peaks is barely observable using an impulse, a step and a sine. Looking at the frequency response, that peak is only 1% of the low frequent gain so it is obvious that it can not be caught, definitely not with contaminating the output with noise. A solution would be to compare the models in the frequency domain. This research has been continued by G. Naus [8].

In the next chapter results of controller design with GP are described. The only difference with system identification is the fitness-function. Its composition will be described in detail.

7 GP & controller design

7.1 Introduction

Application like pick-and-place units, robot arms, printers etc. are created to perform specific tasks. The movable part is driven by a motor and the position is measured. The field of motion control practises the realtime creation of the motor power supply according to measurements, considering user requirements and considering knowledge about the system [5].

The suggested applications above, all have slightly different requirements. A pick-and-place unit should stand still at the desired place as fast as possible. A robot arm must follow a trajectory as good as possible at every time. And a printer should follow a trajectory with a steady velocity. The power of this implementation of the GP algorithm is that arbitrary requirements can be specified in time- and frequency-domain. Because the GP program only optimizes dynamical models with a fixed number of inputs, only the fitness-function has to be changed to make the step from system identification to controller design.

7.2 Fitness-function

Creating a fitness-function for controller design is a difficult task. Each model has to be judged properly, else GP will create wrong controllers with an apparently good fitness. GP optimizes what the user specifies, no more, no less. So the user has to consider carefully what he really desires of a controller without using assumptions valid for PID-like controllers. For example, the gain increases with decreasing frequency in the low-frequency region because of an I-action. For a randomly generated controller containing lots of transfer function blocks this may definitely not be assumed. A dip somewhere in the low-frequency gain could lead to bad tracking performance or bad disturbance rejection.

Dynamical systems

Before presenting all elements of the fitness-function a brief introduction is given about the dynamics used for controller design. The scheme of a closed loop system is shown in Figure 7.1.

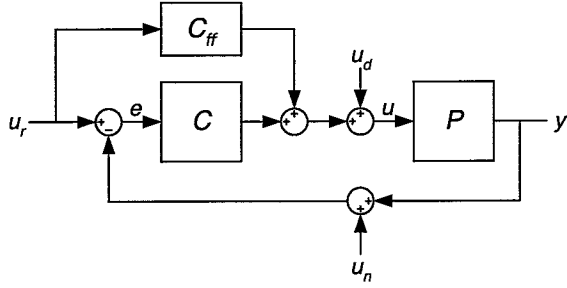


Figure 7.1: Closed loop system.

The closed loop transfer function is given in Equation 7.1 (for a linear SISO system). It shows the relation between the user-defined trajectory and the actual output of the plant. The controller has been divided into a feedback and a feedforward part. Often, both parts are designed separately. The GP program will be offered the error, the velocity and the acceleration signal to create the controller.

$$T(s) = \frac{y(s)}{u_r(s)} = \frac{C(s)P(s)}{1 + C(s)P(s)} \quad (7.1)$$

The open loop transfer function is given in Equation 7.2. It represents the relation between the error e and the plant output y .

$$\frac{y(s)}{e(s)} = C(s)P(s) \quad (7.2)$$

The open loop transfer function plotted in the complex plane is called a Nyquist plot. It is useful to determine stability (passing side of point -1) and robustness (distance to point -1).

A plot of the absolute value and the angle of a transfer function is called a Bode plot. Besides the stability and robustness (gain- and phase-margin), also the bandwidth can be determined. The bandwidth is defined as the frequency where the open loop crosses the 0 dB line.

The sensitivity function is shown in Equation 7.3. It represents the suppression of the disturbance signal $u_d(s)$ effected on the plant input signal $u(s)$. Further, the reciprocal of the absolute value is the distance to the point -1 in a Nyquist plot.

$$S(s) = \frac{u(s)}{u_d(s)} = \frac{1}{1 + C(s)P(s)} \quad (7.3)$$

Performance

Direct performance requirements are time-domain based, like settling time, overshoot, static error etc. Since the GP program can handle time- and frequency-domain criteria, the performance is specified in the time-domain.

The fitness-function is based on a pick-and-place-like task. For the trajectory u_r a 3th order set-point is chosen (see Figure 7.2). T_c is the time that the trajectory becomes constant again. T is the total simulation time. T_s is the settling time, which is specified as the time after which the difference between the plant output and the trajectory stays within an error band until the end of the simulation time T . e_{max} is the maximum absolute error between T_c and T .

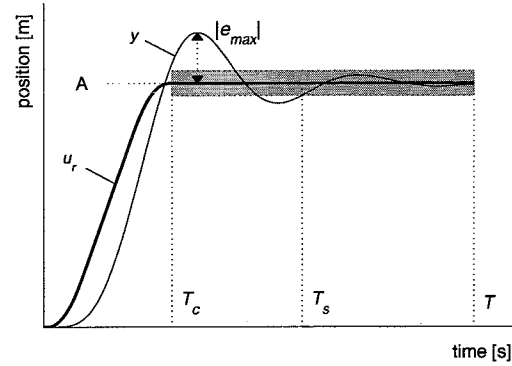


Figure 7.2: Interpretation of a time-domain simulation.

$$\begin{aligned} \text{if } T_s > T_c & \text{ then } f \rightarrow f + \frac{T_s}{T} \\ \text{if } T_s > 0,9T & \text{ then } f \rightarrow f + \frac{e_{max}}{A} \end{aligned} \quad (7.4)$$

The share of the performance criterion to the fitness-function is shown in Equation 7.4. The first line adds the scaled settling time to the fitness f . The second line is added to add more nuance to the fitness-function for less good models. Minimizing this fitness results in an optimum settling time. The way the fitness landscape is shaped, determines the evolution capabilities of GP.

Stability and robustness

Aside from the performance criterion, there are some more criteria desirable, like a stability/robustness criterion and the actuator limit. Both criteria are implemented as a constraint. If a constraint is exceeded, a penalty of $1 \cdot 10^5$ will be added to the fitness. The robustness criterion is included as limiting the maximum sensitivity to 6 dB. This prevents that the open loop does not come too close to the point -1 in a Nyquist plot. This criterion solely is not enough to guarantee stability, because the line in a Nyquist plot could pass the point -1 on the wrong side. This would result in an unstable closed loop behavior. Together with the previous time-domain performance criterion stability is guaranteed, because such an unstable model would blow up and be caught by the performance criterion.

Actuator limit

Every actuator can provide a finite amount of power. Limiting the plant input is needed to prevent creation of unrealistic controllers with too high gains. Applying such controllers would result in performance and stability problems.

Disturbance rejection

With the above criteria GP is able to create controllers which are optimized considering performance, robustness and the actuator limit. The only thing left which is not considered is disturbance rejection. The sensitivity transfer function is a measure for disturbance rejection. This can be added in several ways:

- Decreasing the sensitivity at low frequencies will improve disturbance rejection in that region. This can be added in the fitness-function, e.g. by stimulating controller gain at low frequencies.
- Stimulation of a higher bandwidth causes a larger frequency range to have an improved disturbance rejection. This can be done by adding the term $e^{-\text{bandwidth}}$.
- Inserting a disturbance signal will stimulate disturbance rejection, because GP optimizes the controller to be as fast as possible

within the specified error band. If the error is too big because of the disturbance signal, the only possibility is to suppress it by increasing the controller gain. This is immediately a problem for the optimization process. If other criteria in the fitness-function do not allow just that, then GP is unable to optimize the settling time which would be devastating for the whole optimization process.

Adding white noise is not smart, because a controller can not provide a good disturbance rejection at all frequencies. Improvement of the disturbance rejection at the lower frequencies (below the bandwidth) always results in a worse disturbance rejection in the higher frequencies (above the bandwidth). Better would be to insert low-frequency noise instead of white noise to stimulate the disturbance rejection at the lower frequencies.

Adding noise to the plant input will also prevent that GP creates a pure feedforward controller. The added disturbance signal can be seen as a model uncertainty. However, the intensity of the noise may not be too high causing a bigger error than the error band. The settling time can not be optimized in that case.

Error handling

Finally, erroneous controllers (controllers which cause the closed loop to be unstable and will often cancel the simulation) get the maximum penalty of $1 \cdot 10^{10}$. Controllers with stiff differential equations often consume a lot of simulation time. After a user-specified time the simulation will be aborted and the fitness-function will return the maximum penalty. Some simulations throw an error before this time limit because of unsolvable differential equations. This is also intercepted and the maximum penalty is returned.

The total fitness-function consists of the next elements:

- if $T_s > T_c$ then $f = f + \frac{T_s}{T_c}$
- if $T_s > 0,9T$ then $f = f + \frac{e_{max}}{A}$

- if the maximum sensitivity > 6 dB then $f = f + 10^5$
- if the actuator limit has been exceeded then $f = f + 10^5$
- if the simulation time is too long then $f = 10^{10}$
- if a simulation error occurs then $f = 10^{10}$

7.3 Results

Some simple systems have been taken to let GP create controllers for. This is done to test which combination of elements belongs in the fitness-function. A Pentium IV 2,4 GHz with 256 MB RAM is used for performing the GP runs. The GP settings are equal to the settings used for system identification in Chapter 6 except for the population size which is set to 500 to have a more diverse population. The other difference is that there are three inputs available instead of one.

7.3.1 Controller design for a mass-damper system

System

A mass-damper system is examined which originates from a real setup, a pick-and-place unit placed at an air-bearing. The transfer function of the system is written in Equation 7.5 and the bode plot is shown in Figure 7.3.

$$P(s) = \frac{2,6}{2,6s^2 + 19s} e^{-0,0006s} \quad (7.5)$$

The system has got an actuator limit of 63 N. Considering the purpose of the real setup, a displacement of 0,3 m has to be made using a third order trajectory with a final accuracy of 5 μm . The maxima of the jerk, acceleration and velocity of the trajectory are respectively 1000 m/s^3 , 6,3 m/s^2 and 0,8 m/s . For these conditions a controller has to be designed.

Settings

A time of 2 seconds will be simulated with a sampling frequency of 5 kHz. This time should be long enough so that GP can optimize the settling time. An extra element has been added to

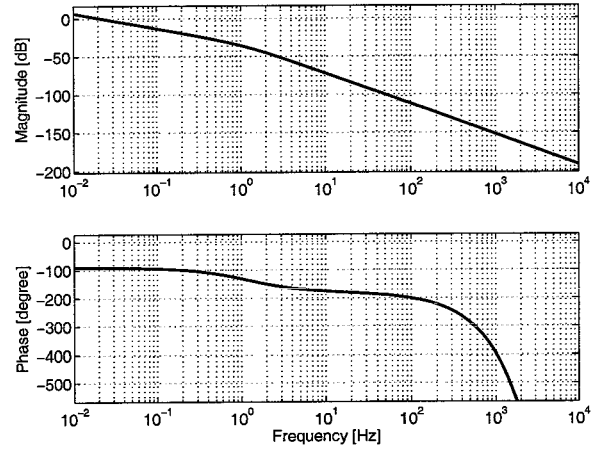


Figure 7.3: Transfer function of a mass-damper system with time delay.

the fitness-function: $f = f + e^{-\text{bandwidth}}$, which stimulates a higher bandwidth. The provided spec was relatively easy to reach. Without the bandwidth-criterion the fitness would be zero and the controller would not be optimized anymore, although the controller could still be optimized a lot with respect to disturbance rejection.

Results

The main characteristics of the produced controller are listed below.

Simulation time	4,9 days
Fitness	2,3e-069
Settling time ($T_s - T_c$)	0 s
Bandwidth	158 Hz
Maximum sensitivity	5,9 dB
Maximum controller output	12 N

GP returned a Simulink model with 2 transfer function blocks, 5 gain blocks, 5 subtract blocks and 6 addition blocks. It uses the error signal, the velocity signal and the acceleration signal so GP has created a feedback + velocity/acceleration feedforward controller. In Figure 7.4 the transfer function of the feedback part is shown.

When the feedforward part is applied solely, the output moves about halfway the trajectory.

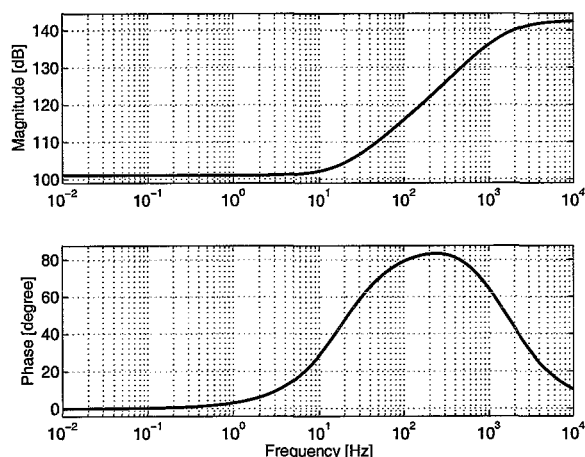
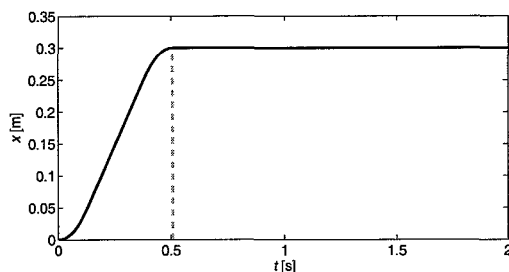
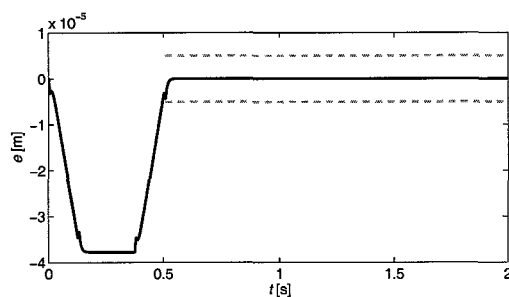


Figure 7.4: Controller transfer function.

Figure 7.5(a), 7.5(b) show that the trajectory is tracked good. The output is within the spec immediately at the constant part of the trajectory.



(a) Time response of output and trajectory



(b) Error between output and trajectory

Figure 7.5: The output compared with the trajectory

Figure 7.6 shows that stability and robustness are preserved and in Figure 7.7 the open loop system is plotted with a bandwidth of 158 Hz.

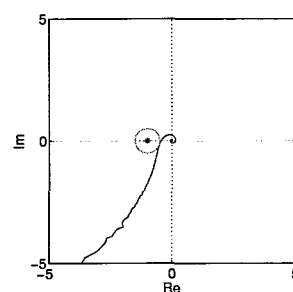


Figure 7.6: Nyquist plot.

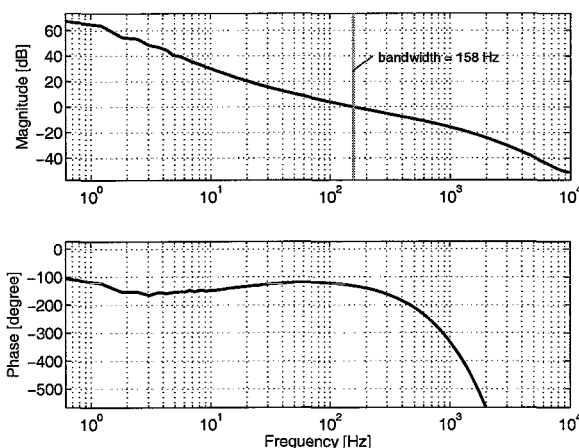


Figure 7.7: Open loop transfer function.

7.3.2 Controller design for a fourth order system

System

The second system is the Pato-setup (the smaller type) located in the Lab of Control Systems Technology at the University of Eindhoven. It consists of two rotating masses with a flexible beam in between (see Figure 7.8). At one mass an electro-motor is connected and the angular position of both masses are measured by encoders with a resolution of 2000 increments per revolution.

This system behaves mainly as a fourth order system. A controller can be designed for two possible cases, measuring at the motor-side and measuring at the load-side. The transfer function of V/x_{motor} is different than V/x_{load} because of the flexible beam in between. Both transfer functions have been determined by measuring the sensitivity in closed loop. A weak

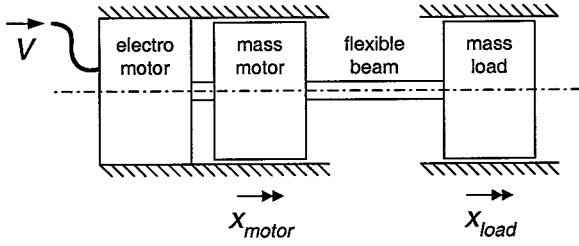


Figure 7.8: Schematic view of the Pato-setup.

PD-controller is used to stabilize the closed loop. Furthermore, a constant speed trajectory is used to avoid nonlinearities like coulomb friction and cogging, which would influence the measurements. Noise is added to the controller output and the motor input is measured. The transfer function of this input-output is called the sensitivity. The system may not change its rotating direction because of this noise, else the nonlinearities will disturb the measurements. Assuming that the implemented controller approximates the theoretical continuous-time controller, the transfer function of the plant is determined by calculating $P = C^{-1}(S^{-1} - 1)$ (see Equation 7.3). A sampling frequency of 500 Hz is used.

In Figure 7.9 the transfer function is plotted measured at the motor-side (grey line). The black line is a fit of the transfer function which is shown in Equation 7.6.

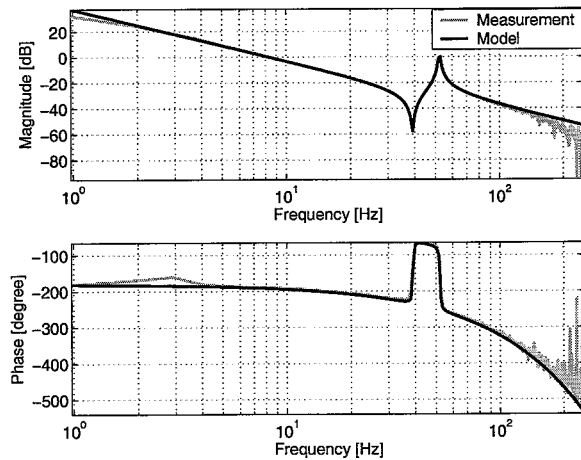


Figure 7.9: Transfer function fit of a Pato-setup, measured at the motor.

$$P(s) = \frac{5000s^2 + 1,47 \cdot 10^4 s + 3,002 \cdot 10^8}{s^4 + 3,943s^3 + 1,08 \cdot 10^5 s^2} e^{-0,004s} \quad (7.6)$$

In Figure 7.10 the transfer function is plotted measured at the load-side (grey line). The black line is a fit of the transfer function which is shown in Equation 7.7.

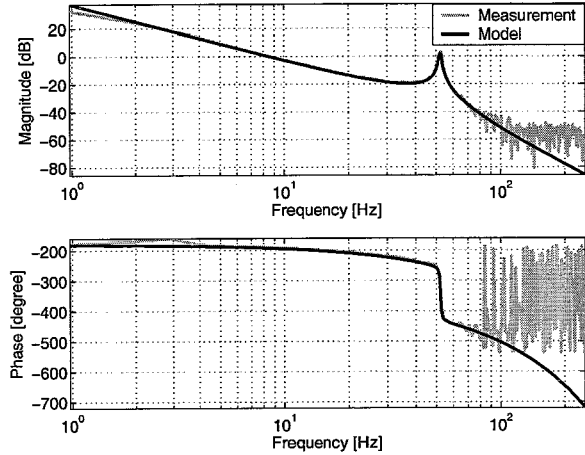


Figure 7.10: Transfer function fit of a Pato-setup, measured at the load.

$$P(s) = \frac{3,002 \cdot 10^8}{s^4 + 3,943s^3 + 1,08 \cdot 10^5 s^2} e^{-0,004s} \quad (7.7)$$

Both systems have a time delay of 4 ms, this is because of the use of a DA converter and the time needed to compute the controller output. This is clearly visible in the transfer functions and limits the performance to control the system. Furthermore, measuring at the load-side has an extra bandwidth limitation, it jumps from -180° to -360° . Both models have been given to the GP program to create a controller for it.

Settings

The objective is to move as fast as possible to another angular position with a maximum error of 0,1%. A third order set-point is used. A time interval of 2 seconds will be simulated with a sampling frequency of 2 kHz. The actuator limit is set to 2 V.

Results of measuring at the motor-side

GP returned a Simulink model with 8 transfer function blocks, 4 gain blocks and 2 subtract blocks. It uses the error signal and the velocity signal so GP has created a feedback + velocity feedforward controller. In Figure 7.11 the transfer function of the feedback part is shown.

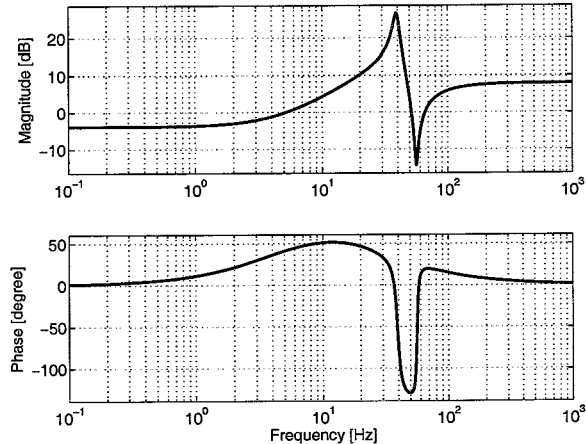


Figure 7.11: Controller transfer function.

The resulting characteristics of the controller are listed below.

Simulation time	2,0 days
Fitness	0,12
Settling time ($T_s - T_c$)	0,056 s
Bandwidth	11 Hz
Maximum sensitivity	5,8 dB
Maximum controller output	0,52 V

Figure 7.12 shows that an overshoot of 8% occurs. No penalty was included in the fitness-function so if a better settling time can be obtained with such an overshoot, GP will do just that.

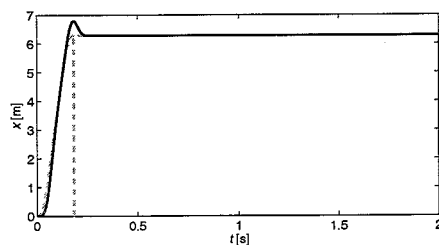


Figure 7.12: Time response.

Figure 7.13 shows that stability and robustness are preserved.

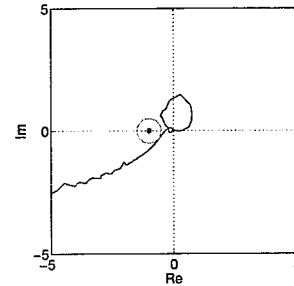


Figure 7.13: Nyquist plot.

The bandwidth can be determined from Figure 7.14. It also shows that the anti-resonance/resonance peak of the plant is suppressed by a slant notch visible in Figure 7.11. Keep in mind that only the settling time is optimized. The notch suppresses the disturbance of the load at the motor. It is obvious that suppressing this disturbance reduces the settling time of the first mass. It is nice to see that GP came up with this without explicitly telling it to do that.

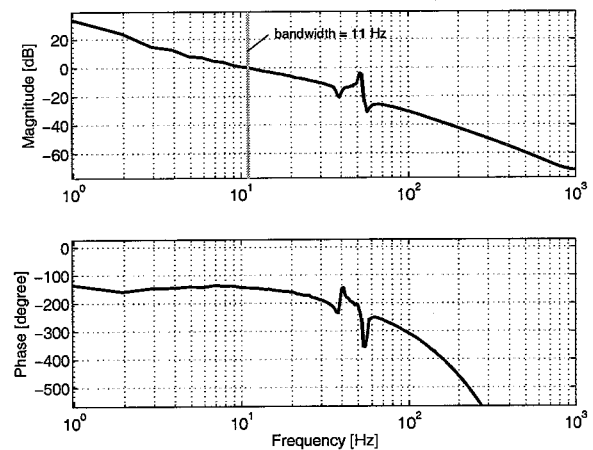


Figure 7.14: Open loop transfer function.

Results of measuring at the load-side

GP returned a Simulink model with 3 transfer function blocks, 1 integrator block, 2 subtract blocks and 1 addition block. It uses only the error signal so GP has created a feedback controller. In Figure 7.15 the transfer function of the feedback part is shown.

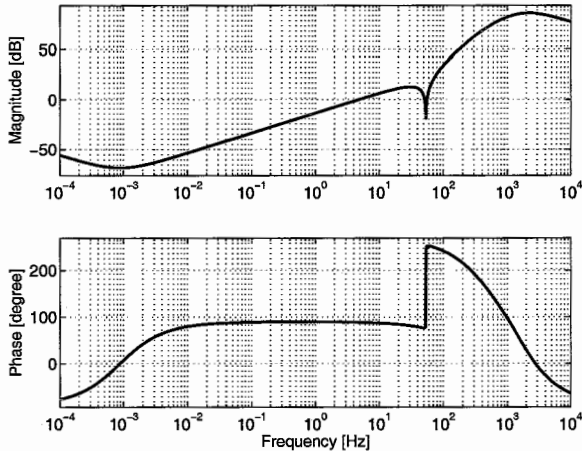


Figure 7.15: Controller transfer function.

GP has created a PID controller with a low-pass filter and a notch which compensates the resonance peak. The characteristics of the controller are listed below.

Simulation time	3,1 days
Fitness	0,076
Settling time ($T_s - T_c$)	0,009 s
Bandwidth	15 Hz
Maximum sensitivity	5,9 dB
Maximum controller output	1.7 V

In Figure 7.16 the response of the output is shown together with the trajectory. During movement of the trajectory, the output lags little bit behind the trajectory. The output does not have any overshoot. Figure 7.17 shows that stability and robustness are preserved. The open loop system is drawn in Figure 7.18.

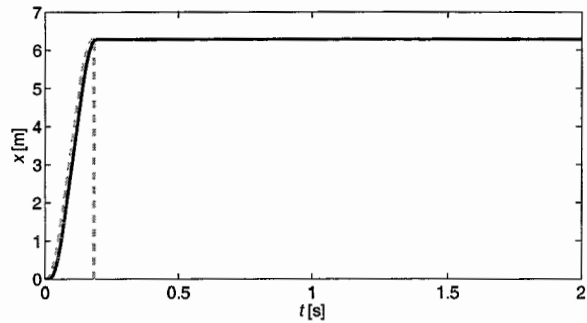


Figure 7.16: Time response.

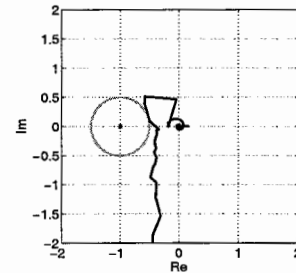


Figure 7.17: Nyquist plot.

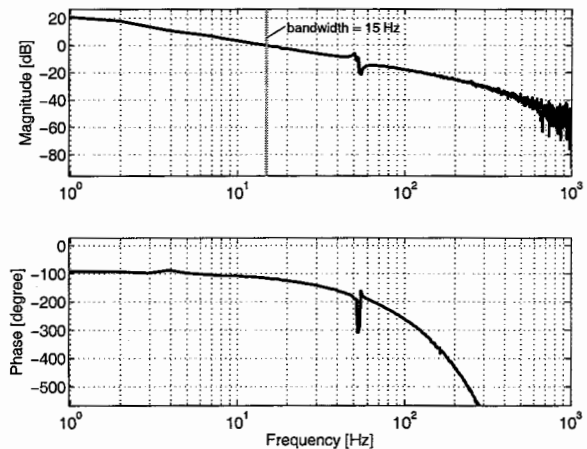


Figure 7.18: Open loop transfer function.

7.4 Summary

Simulations

GP is able to generate controllers which are optimized for the specified performance, stability and robustness. Three simple SISO systems derived from experimental setups were used to create controllers for. Every GP run is different so a variety of controllers have been created. Slight adjustments can be made to the fitness-function settings, so the user is able to balance the importance of various parts.

The mass-damper system used a relative slow trajectory. After a while, GP generates a controller which is immediately within the error band at the constant part of the trajectory. This results in a fitness value of zero, so the GP run is finished by fulfilling all requirements. After this, a criterion to improve disturbance rejection was added to the fitness-function, namely a bandwidth criterion so GP kept on improving the controller.

Furthermore, when only applying the feedforward part, the output of the mass-damper plant came only halfway the prescribed setpoint. It is not fully optimized because GP has created a feedback and feedforward controller simultaneously. Separating these parts in the fitness-function could be considered in future research.

Several attempts to incorporate disturbance rejection have been tried, but no good result was obtained. When inserting white noise or low-frequent noise, GP was not able to optimize the settling time, because the error band could not be reached. Reducing the amplitude of the noise did not work, because then it did not have any effect. The produced controllers were worse than without adding a disturbance signal. The bandwidth criterion only works if GP is able to fulfil all other requirements so this is also no good general criterion. A recommendation for further research is that the user should constrain the sensitivity or the controller gain for a chosen frequency to incorporate disturbance rejection.

The fitness-function is not yet well defined to produce controllers which can be implemented immediately on a real setup, because during the

tests of GP, sensor noise was not included, non-linear effects like coulomb friction were not included and the discretisation on a real setup is not taken into account. This leads to controllers that emphasize the sensor noise too much and sometimes place poles and zeros above half the sampling frequency resulting in aliasing. During tests on a real setup, this was observed. For future research, this should also be penalized in the fitness-function.

In the fitness-function the sensitivity function is determined numerically. However, for linear systems it is also possible to determine the sensitivity function analytically using the "linmod"-command in Matlab. This would improve speed and accuracy. This was not done, because some evaluated plants already contained a non-linear element, namely a time delay block. When using "linmod", this time delay would not be included in the analytical model.

Idea's for future research

One type of controller specifications was used, namely: follow a 3th order trajectory and be as fast as possible within a certain error band at the constant part of the trajectory. Furthermore, stability and robustness have to be preserved. Such a demand is typical for a pick-and-place unit. Other types, like following a constant speed trajectory (printer) or following an arbitrary trajectory (robot-arm) as good as possible, could be examined.

Also other plants which are more complex (higher order, nonlinearities etc...) can be considered. This should be done to test the performance of the GP program and where necessary, adjustments should be made to the fitness-function to improve its generality.

In this thesis, all models are continuous-time models. An interesting variation is to define all models as discrete models. An advantage is that a discrete controller is more close to the implementation of it on a real setup. Furthermore, GP performs better with elementary blocks.

8 Conclusion and recommendations

A GP program which optimizes a dynamical model according to user-specified requirements has been implemented successfully. It has been tested by performing system identification and controller design with it for a few simple systems. Some promising results have been achieved by the GP program. It can be concluded that Genetic Programming, in combination with a numerical optimization method for the parameters of the GP model, is suitable for controller design. In this thesis a Genetic Algorithm has been used to optimize the parameters of a GP model.

The application is still under development. Improvements have been achieved by fixing some minor bugs, by tuning the GP-algorithm for better convergence to a good solution and most important, by defining a suitable fitness-function. The fitness-function represents the user's requirements. For controller design the main items are stability, performance and robustness. All parts of the application have to be constructed properly else the chance to get a good result decreases drastically.

Recommendations

Until now only controllers have been designed for a second and a fourth order system including time delay. This research can be continued by refining the fitness-function, e.g. incorporating discretisation such that the controller will perform well on an experimental setup. Also some more complex systems should be tested to explore the generality of the fitness-function. We think of systems with coulomb friction and higher order systems. The fitness-function should be defined in such a way that it can reflect the control objectives for a wide range of systems.

Another possible implementation is to design controllers in the discrete z -domain using a network structure. This has some advantages with respect to the continuous s -domain. Blocks like TF1 and TF2 are not needed anymore. These can be constructed using elementary blocks. As

seen before, GP prefers the use of elementary blocks so the genetic operations can operate better. Furthermore, the design of a discrete controller is closer to the implementation on an experimental setup. It does not have to be discretized like continuous controllers. A disadvantage is that the result is harder to interpret.

GP's main task is optimizing a structure of blocks. A dynamical model can contain a lot of parameters. These parameters are randomly created. Because GP is not very good at the optimization of those parameters it must be assisted by a dedicated algorithm for optimizing the parameters of a model. In this thesis a GA is used to perform this task. The performance of this algorithm is crucial to obtain a good result within reasonable computing time. The parameters should first be optimized to be able to optimize the structure. This is very time consuming and every gain at this is desirable. Another optimization algorithm could be tested to compare it with GA. A suitable candidate is Simulated Annealing combined with the Nelder-Mead Simplex method. In literature it has achieved some promising results, which indicates that it also could be useful for our problem.

In this research only linear system identification and linear controller design have been explored. The GP program is the same for both. It is a parametric and structural optimization algorithm for (non)linear dynamical models. The only difference is the fitness-function. Other fields of application are:

- Nonlinear control, e.g. gain scheduling.
- Identification of nonlinear systems.
- Signal processing, e.g. synthesis of dedicated filters.

List of Figures

2.1	An example of a tree structure, $y = ue^u + u^2 - 2$.	7
2.2	An overview of the GP algorithm.	8
2.3	An illustration of a crossover operation.	9
2.4	An illustration of a mutation operation.	9
2.5	An illustration of a reproduction operation.	10
3.1	Overview of GP research.	10
4.1	A global view of the architecture of the implemented GP algorithm.	12
4.2	Properties of a second order transfer function.	14
6.1	Plant to be identified.	19
6.2	Best GP model.	19
6.3	Comparison of the plant and the best model using the same input data as used in the fitness-function.	20
6.4	Validation of the best model using different input data.	20
6.5	Plant to be identified.	20
6.6	Best GP model.	21
6.7	Comparison of the plant and the best model using the same input data as in the fitness-function.	21
6.8	Validation of the best model using different input data.	21
6.9	Transfer function (gain, phase) and coherence function	21
7.1	Closed loop system.	23
7.2	Interpretation of a time-domain simulation.	23
7.3	Transfer function of a mass-damper system with time delay.	25
7.4	Controller transfer function.	26
7.5	The output compared with the trajectory	26
7.6	Nyquist plot.	26
7.7	Open loop transfer function.	26
7.8	Schematic view of the Pato-setup.	27
7.9	Transfer function fit of a Pato-setup, measured at the motor.	27
7.10	Transfer function fit of a Pato-setup, measured at the load.	27
7.11	Controller transfer function.	28
7.12	Time response.	28
7.13	Nyquist plot.	28
7.14	Open loop transfer function.	28
7.15	Controller transfer function.	29
7.16	Time response.	29
7.17	Nyquist plot.	29
7.18	Open loop transfer function.	29

List of Symbols

Symbol	Unit	Description
A	[m]	Amplitude of third order setpoint
C	[N/m]	Feedback controller model
C_{ff}	[N/m]	Feedforward controller model
e	[m]	Error signal
e_{max}	[m]	Maximum error
f	[-]	Fitness value
m	[-]	Amount of inputs
n	[-]	Amount of data points
n	[-]	Maximum initialization value
P	[m/N]	Plant model
s	[rad/s]	Complex variable used by Laplace transforms
S	[-]	Sensitivity
t	[s]	Time
T	[-]	Complementary sensitivity or closed loop transfer function
T	[s]	Total simulation time
T_c	[s]	End of third order setpoint
T_s	[s]	Settling time
u	[m]	Plant input signal
u_d	[m]	Disturbance signal
u_n	[m]	Sensor noise signal
u_r	[m]	Reference signal
w	[-]	Weighting factor
x	[m]	Position
y	[m]	Plant output signal
y_m	[m]	Output data of a simulation of a Simulink model
y_p	[m]	Output data of a simulation of a plant

Abbreviations

AD	Analog Digital
ADF	Automatically defined function
CFT	Centre For Industrial Technology
DA	Digital Analog
EA	Evolutionary Algorithms
EO	Evolving Objects
ES	Evolutionary Strategies
GA	Genetic Algorithm
GUI	Graphical User Interface
ITAE	Integral of Time-weighted Absolute Error
m-file	Matlab file
MIMO	Multiple Output Multiple Input
ML	Machine Learning
NN	Neural Networks
PID	Proportional, Integral, Differential
S-function	Simulink function
SA	Simulated Annealing
SISO	Single Output Single Input

References

- [1] A. Fraser, "Genetic Programming in C++", University Of Salford, 1994
- [2] D. Bruijnen, "Cdrom of Automated Controller Design using GP", 2003
- [3] D. Hanselman, B. Littlefield, *Mastering Matlab 6, A Comprehensive Tutorial and Reference*, Prentice Hall inc., 2001
- [4] D. Vangheluwe, *Genetic Algorithm applied to Parameter Design, A Case study on a Reliability Problem*
- [5] G. Franklin, J. Powell, A. Emami-Naeini, *Feedback Control Of Dynamical Systems, third edition*, Addison-Wesley Publishing Company, inc., 1994
- [6] G. Gray, Y. Li, D. Murray-Smith, K. Sharman, "Structural system identification using genetic programming and a block diagram oriented simulation tool", 1996
- [7] J. Dabney, T. Harman, *Mastering Simulink 4*, Prentice Hall inc., 2001
- [8] G. Naus, "System Identification using GP", 2003
- [9] <http://www.revolutionaryengineering.com/EA1.html>
- [10] <http://eodev.sourceforge.net>
- [11] <http://www.geneticprogramming.com>
- [12] <http://www.genetic-programming.org>
- [13] <http://www.ieee.org>
- [14] <http://asd.bbn.com/papers/traffic/traffic.html>
- [15] I. Soute, "Genetic Programming", TU Eindhoven mechanical engineering, Dynamic Systems Design, report number 2000.25, september 2000.
- [16] J. Koza, M. Keane, F. Bennett, J. Yu, W. Mydlowec, O. Stiffelman, *Automatic Creation of both the Topology and Parameters for a Robust Controller by Means of Genetic Programming*, 1999
- [17] J. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*, MIT press, Cambridge, MA, 1992.
- [18] M. Keijzer, J.J. Merelo, G. Romero, M. Schoenauer, "Evolving Objects: a general purpose evolutionary computation library", 2001
- [19] M. Keijzer, "EO Reference Manual 0.9.2", 2002
- [20] W. Banzhaf, P. Nordin, R. Keller and F. Francone, *Genetic Programming an Introduction on the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufman Publishers, Inc., 1998.
- [21] W. Langdon, A. Qureshi, "Genetic Programming - Computers using Natural Selection to generate programs", Dept of Computer Science, University College London.
- [22] W. Langdon, R. Poli, "Better Trained Ants for Genetic Programming", 1998
- [23] W. Press, S. Teukolsky, W. Vetterling, B. Flannery, *Numerical recipes in C++, The Art of Scientific Computing Second Edition*, Cambridge University Press, 2002

A S-functions

A.1 calcstop

calcstop.dll is an S-function to be used in a Simulink model. It tracks the simulation time and terminates the simulation if the simulation time is larger than the parameter value of the S-function block. The source code is shown below. It was compiled in Matlab by typing: ">> mex calcstop.c".

```
#define S_FUNCTION_NAME  calcstop
#define S_FUNCTION_LEVEL 2
#include "simstruc.h" #
include "time.h"

time_t begin;
const int request = 1;

static void mdlInitializeSizes(SimStruct *S) {
    ssSetNumSFcnParams(S, 1);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) return;
    if (!ssSetNumInputPorts(S, 0)) return;
    if (!ssSetNumOutputPorts(S, 0)) return;
    ssSetNumSampleTimes(S, 1);
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE |
        SS_OPTION_USE_TLC_WITH_ACCELERATOR);
    begin = time(NULL);
}

static void mdlInitializeSampleTimes(SimStruct *S) { }

static void mdlOutputs(SimStruct *S, int_T tid) {
    time_t now;
    mxArray *y = ssGetSFcnParam(S, 0);
    double *k = mxGetPr(y);
    now = time(NULL);
    if ((now-begin)>*k) ssSetStopRequested(S, request);
}

static void mdlTerminate(SimStruct *S) { }

#ifdef MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

A.2 protecteddiv

protecteddiv.dll is an S-function for Simulink which is called a protected division. It works the same as a normal division except when the denominator approaches zero. The denominator saturates if its absolute value is smaller than 10^{-6} . The sign is unaffected. If the denominator is zero, it is set to $+10^{-6}$. The source code of this S-function is shown below.

```
#define S_FUNCTION_NAME protecteddiv
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

static void mdlInitializeSizes(SimStruct *S) {
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    if (!ssSetNumInputPorts(S, 2)) return;
    ssSetInputPortWidth(S, 0, 1);
    ssSetInputPortWidth(S, 1, 1);
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 1, 1);

    if (!ssSetNumOutputPorts(S, 1)) return;
    ssSetOutputPortWidth(S, 0, 1);

    ssSetNumSampleTimes(S, 1);

    /* Take care when specifying exception free code - see sfuntmpl_doc.c */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE | SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}

static void mdlInitializeSampleTimes(SimStruct *S) {
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

static void mdlOutputs(SimStruct *S, int_T tid) {
    InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S, 0);
    real_T *y = ssGetOutputPortRealSignal(S, 0);

    if ((*uPtrs[1] >= 0) && (*uPtrs[1] < 1e-6)) *y = *uPtrs[0] * 1e6;
    else if ((*uPtrs[1] >= 1e-6) && (*uPtrs[1] < 0)) *y = *uPtrs[0] * -1e6;
    else *y = *uPtrs[0] / *uPtrs[1];
}

static void mdlTerminate(SimStruct *S) {}

#ifdef MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

B Fitness-function m-file

The fitness-function returns a value which indicates how good a GP model satisfies user-specified requirements. It is built into one file, so all elements of the fitness-function are hold together. This ensures that the fitness-function is the same as the fitness-function used during the GP run. Enough info-lines are added to explain what is done in the fitness-function. A suggestion for further research is to subdivide this fitness-function because it has grown quite large which is not very user friendly. The only part to be edited by the user considering controller design is part 2. Another suggestion is to make a GUI to set all parameters of the fitness-function and set the settings of GP. Then you do not have to edit the fitness-function by hand anymore.

The fitness-function can be divided into the following parts:

1. **function description:** This part describes the usage of the fitness-function by the GP program and by the user.
2. **adjustable parameters:** These parameters can be adjusted to specify your demands. The rest of the fitness-function does not need to be changed for controller design.
3. **pre-checks:** These checks are inserted for the user's convenience. It makes the fitness-function more flexible. For example, if you want to evaluated the best model so far during a GP run, you can simply type `fitness('temp_model','mdls')` which evaluates the best model, without having to look what the most recent model is.
4. **setup simulation model:** This part inserts the GP model in another Simulink-model which can then be simulated.
5. **fitness determination:** Here, simulation are run and the fitness value is determined.
6. **validity check of the fitness value:** After the fitness-function has been determined, a check is made if the fitness-value is valid. The fitness-function should always return a vaild fitness value!
7. **visualisation of the results:** If a directory name was specified the results of the fitness evaluation are plotted. Because the GP program never specifies the directory name (it uses the GP model in the memory and files in the same directory as the fitness-function), the results are not shown if the GP program evaluates the fitness-function. This is desired because hundred of thousands of evaluations are done. We are only interested in the best models.

Fitness-function for controller design

The fitness m-file is shown next. The previously described parts are also indicated.

— 1. function description —

```
function [f, system_model] = fitness(gp_model, dirname)
%FITNESS returns a fitness value which indicates the performance of
%a simulink-model according to user-specified requirements.
%usage by GP: f = FITNESS('gp_model');
%           [f, system_model] = FITNESS;
%
%usage by user: f = FITNESS('gp_model','dirname');
%
% gp_model: A simulink-model with the name: <gp_model>.mdl
%           It contains m inputs and 1 output block and
%           can be used by the command SIM:
%
%           [T,X,Y] = sim('model',t,[],[t u]);
%
%           inputs
%           t: timevector, n-by-1 matrix
%           u: inputvector, n-by-m matrix
%           outputs
%           T: timevector, n-by-1 matrix
%           X: respons of all states in 'model', n-by-? matrix
%           Y: outputvector, n-by-1 matrix
%
%           If gp_model is set to 'temp_model', the most recent numbered file
%           'temp_modelxxx.mdl' will be selected if it exists. Useful for
%           displaying intermediate results during a GP run.
%
% dirname: (optional) the gp_model file will be search in
%           this directory, also the input-output data is plotted.
%
% f: positive real value which indicates the performance
%    of a simulink model. A larger value of f is equal to
%    a worse performance.
%
% system_model: (optional) This will return the system_model name without
%                simulating anything
%
% Koninklijke Philips Electronics N.V.
% Centre for Industrial Technology (CFT)
% Created: 01-2003
% By:      DJH Bruijnen
```

— 2. adjustable parameters —

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PARAMETERS TO BE EDITED BY THE USER
```

%system parameters	
system_model = 'pato_load';	%user-specified plant, must be in the same directory!
s_out.PlantOut = 1;	%outport at plant output
s_out.ControllerOut = 2;	%outport at controller output
s_in.PlantOut = 1;	%inport with sensor noise signal
s_in.ControllerOut = 2;	%inport with disturbance signal
s_in.RefAcc = 3;	%inport with reference acceleration signal
s_in.RefVel = 4;	%inport with reference velocity signal
s_in.RefPos = 5;	%inport with reference position signal
s_in.Amount = length(struct2cell(s_in));	%amount of inports of the system
%controller parameters	
c_out.ControllerOut = 1;	%outport at controller output
c_in.RefAcc = 1;	%inport with reference acceleration signal
c_in.RefVel = 2;	%inport with reference velocity signal
c_in.PosError = 3;	%inport with position error signal
c_in.Amount = length(struct2cell(c_in));	%amount of inports of the controller

```

%trajectory parameters
dt = .002;           %sample time for 3rd order trajectory
T = 2;               %simulation time
A = 2*pi;            %amplitude of trajectory
jmax = 100000;        %maximum jerk of trajectory in m/s3
amax = 1000;          %maximum acceleration of trajectory in m/s2
vmax = 50;            %maximum velocity of trajectory in m/s

%frequency analysis parameters
sens.n = 2^11;        %Amount of data points for determining the sensitivity
sens.parts = 2;        %Division of total signal in smaller parts
sens.overlap = 0.5;    %Overlap of smaller parts
fs = 1000;            %sampling frequency

%fitness penalty parameters
BadSimPenalty = 1e10;  %Penalty for incomplete simulations
ConstraintPenalty = 1e5; %Penalty if a constraint has been exceeded

%requirement parameters
accuracy = .001;       %settling time is determined using a max. error of accuracy*100%
SensLimit = 6;          %constraint: maximum sensitivity in dB
ActuatorLimit = 1;      %constraint: maximum actuator amplitude in N
MST = 10;               %constraint: maximum simulation time in s
disturb = 0;            %amplitude of disturbance signal (noise)
noise = 0;               %amplitude of sensor noise signal (noise)
bandwidth_factor = 0;    %bandwidth_factor = 0: disabled else f = f*exp(-bandwidth_factor*bandwidth)

```

— 3. pre-checks —

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% THE REST OF THE M-FILE DOES NOT HAVE TO BE CHANGED FOR CONTROLLER DESIGN!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

%%%%%%%%%%%%%%
% PRE-CHECKS

```

```

%this part is only used by gp4so. It is used to get the system_model
%without simulating it
if nargin == 2
    f = BadSimPenalty;
    return
end

%check if another dir is requested
dir_exists = exist('dirname');
if dir_exists
    %set path
    current_dir = cd;
    addpath(current_dir); %necessary! enables the use of the plant and dll's
    cd(dirname);

    %check fitness-data-file
    if exist([cd '\ gp_model '_fitness.log'])
        fitness_plot_data = load([cd '\ gp_model '_fitness.log']);
    elseif exist([cd '\fitness.log'])
        fitness_plot_data = load([cd '\fitness.log']);
    else
        fitness_plot_data = 0;
    end

    %check if there is a plant saved during finalization of a GP run
    if exist([cd '\ gp_model '_used_system.mdl'])
        system_model = [gp_model '_used_system']; %use plant of saved result of a GP run
    end

    %check if gp_model=='temp_model', the most recent numbered gp_model will be selected
    if isequal(gp_model,'temp_model')
        D=dir('temp_model*.mdl');
    end

```

```

    D={D.name};
    if exist([gp_model int2str(length(D)) '.mdl'])
        gp_model = [gp_model int2str(length(D))];
    end
end

%check if the gp_model exists
if not(exist(gp_model))
    %return to the initial directory
    cd(current_dir);
    error('gp_model does not exist!')
end
end
end

```

— 4. setup simulation model —

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GET AND SETUP CONTROLLER

try load_system(gp_model); catch; end
try
    add_block('built-in/Inport',[gp_model '/u1']);
    add_block('built-in/Inport',[gp_model '/u2']);
    add_block('built-in/Inport',[gp_model '/u3']);
    add_block('built-in/Outport',[gp_model '/y']);
    add_line(gp_model,'u1/1','GPmodel/1');
    add_line(gp_model,'u2/1','GPmodel/2');
    add_line(gp_model,'u3/1','GPmodel/3');
    add_line(gp_model,'GPmodel/1','y/1');
    addterms(gp_model);
catch
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% CREATE CLOSED LOOP SYSTEM

try bdclose(system_model); catch; end
try load_system(system_model); catch; end
add_block([gp_model, '/GPmodel'],[system_model '/GPmodel'])
add_line(system_model,'ddu_r/1','GPmodel/1');
add_line(system_model,'du_r/1','GPmodel/2');
add_line(system_model,'e/1','GPmodel/3');
add_line(system_model,'GPmodel/1','D/2');

```

— 5. fitness determination —

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DETERMINING FITNESS

f = 0;

%determine zeroes, poles and gain of the controller (only linear models!)
gp_sys = linmod(gp_model);
[gp_z, gp_p, gp_k] = ss2zp(gp_sys.a, gp_sys.b, gp_sys.c, gp_sys.d, c_in.PosError);

%check stability
if any(gp_z > 0) | any(gp_p > 0)
    f = BadSimPenalty;
else
    %create 3rd order trajectory
    [ur, dur, ddur] = setp_3c(A, vmax, amax, jmax, dt, 0, T);
    [n, m] = size(ur); %determining size of trajectory matrix
    t = [0:n-1]*dt; %time vector (n-by-1 matrix)

    %sensor noise and disturbances
    un = (2*rand(n,m)-1)*noise;
    ud = (2*rand(n,m)-1)*disturb;

```

```

%calculate response of the closed loop
[y, sim_error] = simmodel(system_model, t, [un, ud, ddur, dur, ur], m, MST);
if sim_error
    f = BadSimPenalty;
else
    %determine settling time
    absE = abs(y(:,1)-ur);
    spec = find(absE/A > accuracy);
    if length(spec) == 0
        %within spec all the way
        settling_time = 0;
    elseif spec(end) == n
        settling_time = T;
    else
        %interpolate settling time
        spec = spec(end);
        spec_interp = (absE(spec)-A*accuracy)/(absE(spec)-absE(spec+1));
        settling_time = t(spec)*(1-spec_interp) + t(spec+1)*spec_interp;
    end

    %a scaled settling_time penalty is added to the fitness if the
    %position is not within the spec at the constant part
    constant = find(dur ~= 0);
    constant = constant(end);
    if settling_time > t(constant)
        f = f + settling_time/T;
    end

    %add an extra scaled amplitude penalty if the position is not
    %within the spec at time T
    if settling_time > T * .9
        f = f + max(absE(constant:end))/A;
    end

    %controller output constraint
    maxU = max(abs(y(:,s_out.ControllerOut)));
    if maxU > ActuatorLimit
        f = f + ConstraintPenalty;
    end

    %calculate sensitivity
    [S, sim_error] = EstimTF(system_model, fs, sens.n, sens.n/sens.parts, sens.overlap,...
        s_in.Amount, [s_in.ControllerOut s_out.ControllerOut], MST);
    if sim_error
        f = BadSimPenalty;
    else
        %sensitivity constraint
        maxS = max(20*log10(abs(S)));
        if maxS > SensLimit
            f = f + ConstraintPenalty + maxS - SensLimit;
        end

        %bandwidth optimization
        if bandwidth_factor ~= 0
            CPabs = abs(1./S - 1);
            bw = find(CPabs <= 1);
            if any(bw > 1) & CPabs(1) > 1
                bw = bw(1);
                bw = bw + log(CPabs(bw))/log(CPabs(bw-1)/CPabs(bw)); %interpolation on dB-scale
                bandwidth = (bw-1)/(length(S)-1)*fs/2;
            elseif CPabs(1) > 1
                bandwidth = fs/2;
            else
                bandwidth = 0;
            end
            f = f + exp(-bandwidth_factor*bandwidth);
        end
    end
end
end
end

```

— 6. validity check of the fitness value —

```
%%%%%%%%%%
% FINALIZATION
```

```
%check validity of f
if not(exist('f'))
    f = BadSimPenalty;
elseif not(isfinite(f)) | f > BadSimPenalty | f < 0 | not(imag(f) == 0)
    f = BadSimPenalty;
end
```

— 7. visualisation of the results —

```
%plot input output data if a directory is specified
if dir_exists
    %return to the initial directory
    cd(current_dir)

    %remove GPmodel from closed-loop
    delete_line(system_model,'ddu_r/1','GPmodel/1');
    delete_line(system_model,'du_r/1','GPmodel/2');
    delete_line(system_model,'e/1','GPmodel/3');
    delete_line(system_model,'GPmodel/1','D/2');
    delete_block([system_model '/GPmodel']);

    %determine bandwidth if it isn't calculated before
    if f ~= BadSimPenalty
        CPabs = abs(1./S - 1);
        bw = find(CPabs <= 1);
        if any(bw > 1) & CPabs(1) > 1
            bw = bw(1);
            bw = bw + log(CPabs(bw))/log(CPabs(bw-1)/CPabs(bw)); %interpolation on dB-scale
            bandwidth = (bw-1)/(length(S)-1)*fs/2;
        elseif CPabs(1) > 1
            bandwidth = fs/2;
        else
            bandwidth = 0;
        end
    end

    %plot info to screen
    disp(' Genetic Programming for controller design |')
    disp(' FITNESS-FUNCTION |')
    disp(' Created by DJH Bruijnen |')
    disp(' TUE / Philips CFT |')
    disp(' ')
    disp('---simulink models---')
    disp(['system: ' system_model '.mdl'])
    disp(['gp model: ' gp_model '.mdl'])
    disp(' ')
    disp('---controller---')

    gp_sys = zpk(gp_z, gp_p, gp_k);
    gp_sys = minreal(gp_sys);
    [gp_z, gp_p, gp_k] = zpndata(gp_sys);
    gp_z = gp_z{:};
    gp_p = gp_p{:};

    disp('poles:')
    [real_gp_p, index_gp_p] = sort(real(gp_p));
    disp(gp_p(index_gp_p))
    disp('zeros:')
    [real_gp_z, index_gp_z] = sort(real(gp_z));
    disp(gp_z(index_gp_z))
    disp(['gain: ' num2str(gp_k)])
```

```

disp(' ')
if f ~= BadSimPenalty
    disp('---performance---')
    if settling_time == T
        disp(['settling time: > ' num2str(settling_time) ' s (error: < ' num2str(accuracy*100) '%)'])
    else
        disp(['settling time: ' num2str(settling_time) ' s (error: < ' num2str(accuracy*100) '%)'])
    end
    disp(['bandwidth: ' num2str(bandwidth) ' Hz'])
    disp(' ')
    disp('---constraints---')
    disp(['max. sensitivity: ' num2str(maxS) ' dB (< ' num2str(SensLimit) ' dB)'])
    disp(['max. controller output: ' num2str(maxU) ' N (< ' num2str(ActuatorLimit) ' N)'])
else
    disp('---The gp model is unstable or simulation problems have occurred---')
    return
end

%determining transfer function of plant and controller
[Tplant, Terror] = EstimTF(system_model, fs, sens.n, sens.n/sens.parts, sens.overlap,...
    s_in.Amount, [s_in.ControllerOut s_out.PlantOut], MST);
[Tcontroller, Terror] = EstimTF(gp_model, fs, sens.n, sens.n/sens.parts, sens.overlap,...
    c_in.Amount, [c_in.PosError c_out.ControllerOut], MST);
freq = linspace(0,fs/2,length(S))';

%plot results if the simulation was successful
close all

if not(sim_error)
    figure %FIRST FIGURE

    %ur and y
    subplot(221)
    plot([t(constant) t(constant)],[0 A],':','color',[.7 .7 .7])
    hold on
    plot(t, ur,'color',[.7 .7 .7])
    plot(t, y(:,1),'k')
    axis tight
    title('r and y')
    xlabel('t [s]')
    ylabel('x [m]')

    %y - ur
    subplot(222)
    plot([t(constant); T],[A*accuracy -A*accuracy; A*accuracy -A*accuracy],':','color',[.7,.7,.7])
    hold on
    plot(t, y(:,1)-ur,'k')
    axis tight
    title('error: y - r')
    xlabel('t [s]')
    ylabel('x [m]')

    %plant input
    subplot(223)
    plot(t, y(:,2),'k')
    hold on
    plot([0;T],[ActuatorLimit -ActuatorLimit; ActuatorLimit -ActuatorLimit],':','color',[.7 .7 .7])
    axis tight
    title('plant input')
    xlabel('t [s]')
    ylabel('F [N]')

    %plot fitness data
    subplot(224)
    if not(isequal(fitness_plot_data,0))
        fitness_plot_data(find(fitness_plot_data(:,2) == 0),2) = 1e-15;
        n = size(fitness_plot_data,1);
        x = zeros(2*n-1,1);
        y = zeros(2*n-1,1);

        %setup plot data

```

```

    for i=1:n-1
        x(2*i-1:2*i) = fitness_plot_data(i:i+1,1)-[0;1];
        y(2*i-1:2*i) = fitness_plot_data(i,2);
    end
    x(2*n-1) = fitness_plot_data(n,1);
    y(2*n-1) = fitness_plot_data(n,2);

    %plot results
    miny = min(fitness_plot_data(:,2))/2;
    maxy = max(fitness_plot_data(ceil(n/10):n,2));
    semilogy([fitness_plot_data(2:n-1,1) fitness_plot_data(2:n-1,1)]',...
        [fitness_plot_data(2:n-1,2) miny*ones(n-2,1)]', 'Color', [.7 .7 .7])
    hold on
    semilogy(x,y,'k','LineWidth',3)
    axis([min(x) max(x) miny maxy])
    grid
    xlabel('fitness evaluations')
    ylabel('fitness')
    title('fitness progression')
end

figure %SECOND FIGURE

%open loop gain
subplot(221)
ol_dB = 20*log10(abs(1./S(2:end)-1));
semilogx([bandwidth bandwidth],[min(ol_dB) max(ol_dB)], 'color', [.7 .7 .7])
hold on
semilogx(freq(2:end), ol_dB, 'k')
axis tight
grid
title('open loop gain')
xlabel('f [Hz]')
ylabel('dB')

%open loop phase
subplot(223)
semilogx(freq(2:end), 180/pi*unwrap(angle(1./S(2:end)-1)), 'k')
axis tight
grid
title('open loop phase')
xlabel('f [Hz]')
ylabel('degree')

%nyquist
subplot(222)
plot(real(1./S(2:end)-1), imag(1./S(2:end)-1), 'k', -1, 0, '*k')
hold on
w=linspace(0,2*pi,100);
plot(-1+sin(w)/(10^(SensLimit/20)), cos(w)/(10^(SensLimit/20)), 'color', [.7 .7 .7])
axis([-5 5, -5 5])
grid
title('nyquist')
xlabel('Re')
ylabel('Im')

%sensitivity
subplot(224)
semilogx(freq(2:end), 20*log10(abs(S(2:end))), 'k')
axis tight
grid
title('sensitivity')
xlabel('f [Hz]')
ylabel('dB')

figure %THIRD FIGURE

%plant gain
subplot(231)
semilogx(freq(2:end), 20*log10(abs(Tplant(2:end))), 'k')
axis tight

```

```
grid
title('plant gain')
xlabel('f [Hz]')
ylabel('dB')

%plant phase
subplot(234)
semilogx(freq(2:end), 180/pi*unwrap(angle(Tplant(2:end))), 'k')
axis tight
grid
title('plant phase')
xlabel('f [Hz]')
ylabel('degree')

%controller gain
subplot(232)
semilogx(freq(2:end), 20*log10(abs(Tcontroller(2:end))), 'k')
axis tight
grid
title('controller gain')
xlabel('f [Hz]')
ylabel('dB')

%controller phase
subplot(235)
semilogx(freq(2:end), 180/pi*unwrap(angle(Tcontroller(2:end))), 'k')
axis tight
grid
title('controller phase')
xlabel('f [Hz]')
ylabel('degree')

%closed loop gain
subplot(233)
semilogx(freq(2:end), 20*log10(abs(1-S(2:end))), 'k')
axis tight
grid
title('closed loop gain')
xlabel('f [Hz]')
ylabel('dB')

%closed loop phase
subplot(236)
semilogx(freq(2:end), 180/pi*unwrap(angle(1-S(2:end))), 'k')
axis tight
grid
title('closed loop phase')
xlabel('f [Hz]')
ylabel('degree')
end

%show the closed loop system
open_system([system_model '/plant']);
open_system([gp_model '/GPmodel']);
end
```

C GA implementation

Like GP, GA also uses a population. The major difference is that the structure of an individual is a fixed amount of numbers on a row. The working principle is written next. The words between brackets are parameters of GA which can be specified in a so-called ini-file `gp.ini`.

"PopSize" is the size of the population. Every generation, the best parents are selected with an amount of "Parents". For each new individual an amount of "Recombi" parents (Recombined parents) are randomly selected out of the best parents. A new individual is created by applying a crossover operator at these recombined parents. An amount of "PopSize" new individuals are created.

The standard deviation of the search step is multiplied or divided (equal chance) by "Alpha" for each value of each individual. This enables adjustments to the order of magnitude, because the standard deviation of the search step can get a factor "Alpha" bigger or a factor "Alpha" smaller. A choice of 1,5 for "Alpha" is the default.

A step is taken for each value of each individual by drawing a number from a normal probability distribution, using its new standard deviation of the search step. There is a standard deviation parameter for each value of an individual, because the order of magnitude of those values can vary a lot.

The standard deviation of the initial step is determined by multiplying the absolute value of the parameters in a model by "InitStepSizeFactor". Do not set this value too high else it makes too big jumps in the first generations so it is not able to optimize locally. This is the case when it is set to 1. A value somewhere between 0,01 and 0,1 works fine.

Every generation the best individuals are selected to produce offspring with accompanying step sizes and parameter values. This will go on until the amount of generations "MaxGen" has been reached.

"KeepBest" is set to true if you want GA to return the best individual of a GA optimization when the maximum amount of generations is reached. When set to false, it only returns the best of the last generation. An advantage of this is that GP will not get stuck in a local minimum. In the GP algorithm the current best individual is quite often selected. If GA does not change its values because it is in a local minimum and can not get out, the amount of identical individuals will grow in the GP population. The diversity decreases and it will be impossible to optimize.

So keeping the best of all generations could lead to stagnation of the GP progress resulting in a GP population of all the same individuals with identical parameters.

A more detailed description of the algorithm can be found in [4].

The GA is implemented in `eoGaOperator.h` as a genetic operation of GP. See the GP CDRom [2] to view the source code.

D GP program

D.1 User manual

To become familiar with the software for controller design with GP (gp4cd), first all supplied files are listed. After that, some more detail is given about these files. Finally an instruction list is formulated to perform a GP run successfully considering the user's desires.

D.1.1 Gp4cd-files

Here is a list with all supplied files. They are listed relative to the root-dir of the application. The default of the "simworkdir" is "." and the default of the "modelsavedir" is "mdls".

Files which do not need to be altered are:

- .\gp4so.exe
- .\<simworkdir>\estimtf.m
- .\<simworkdir>\simmodel.m
- .\<simworkdir>\setp_3c.m
- .\<simworkdir>\calcstop.dll
- .\<simworkdir>\protecteddiv.dll

Files which can be altered by the user according to his desires are:

- .\gp.ini
- .\<simworkdir>\fitness m-file (name is specified in gp.ini)

Some working sample files are:

- .\<simworkdir>\fitness.m
- .\<simworkdir>\plant_2nd_order.mdl
- .\<simworkdir>\fitness_pato.m
- .\<simworkdir>\pato_motor.mdl
- .\<simworkdir>\pato_load.mdl

Created files during a GP run are:

- gp_model (a temporary Simulink-model only in memory, which may not be saved!)
- .\<simworkdir>\<modelsavedir>\temp_modelxxx.mdl
- .\<simworkdir>\<modelsavedir>\fitness.log

Created files after a GP run are:

- .\<simworkdir>\<savedir>\<savename>.mdl
- .\<simworkdir>\<savedir>\<savename>_fitness.log
- .\<simworkdir>\<savedir>\<savename>_fitness.m
- .\<simworkdir>\<savedir>\<savename>_used_system.mdl
- .\<simworkdir>\<savedir>\<savename>_logdata.txt
- .\<simworkdir>\loghistory.txt

D.1.2 Description of files

gp4so.exe

This is the executable file which starts a GP run. Before execution, the settings in `gp.ini` should be checked and the fitness-function must be set up correctly. After that, the only thing left is running `gp4so.exe` and waiting for a good solution which minimizes the fitness-function. This can take some hours to a few days. During a GP run you can view intermediate results. The easiest way to do that is to press the Pause-button on the keyboard and then go to the Matlab command window which has been opened by the GP program. Some ways to examine intermediate results are listed next:

- To examine the current model where `gp4so` is working on, type: `fitness('gp_model','.')'`
- To only view the current model, type: `open_system('gp_model/GPmodel')`
- To examine the best model so far, type: `fitness('temp_model','mdls')`
- To examine a model which was the best model in an earlier state, type: `fitness('temp_modelxxx','mdls')`. ("xxx" is the number of that model)

When evaluating a `temp_modelxxx.mdl` also a plot of the fitness progression is drawn. This is only drawn until the best model so far, so not until the present time. You can determine how long ago that best model was found by looking at the amount of fitness evaluations at the end of the current generation. This is useful when considering the convergence of the GP run. Every generation some statistics are shown, like the best fitness in the population, the average fitness of the population, the amount of fitness evaluations and the generation number.

Another possibility for examining intermediate results is using another Matlab command window. Then you do not have to press the Pause-button, however you also can not examine the current model. Only the best models which are saved in the directory "mdls" can be examined.

If you see that the GP run has converged to a certain level or if you just want to abort the GP run, then you can press any key on the keyboard. After a generation is completed you will be asked if you really want to quit. By typing "y", you get the option to save the model by typing the filename and sub-directory. If the file already exists a message appears if you want to overwrite it. Separately from the model other data is saved, so the settings of GP are kept. Also the fitness-function and the system model are saved, so the performance of the generated GP model can be evaluated at any time independently from the current state of the fitness-function and the plant/system model. Finally the fitness-progression is saved. With all this data it is possible to initialize the GP program equally to an earlier state to reproduce the result. For example, if you have saved the model as "model7" in the directory "test" you can examine the model with the original fitness-function and system model as follows:

```
cd test
model7_fitness('model7','.')'
```

If the current fitness-function has not been changed you can simply use: `fitness('model7','test')`. If `model7_used_system.mdl` exists in that directory, it will be used else the system model defined in the fitness-function will be used.

If you want to abort the GP run, you do not want to wait for the whole generation and you do not want to save the best result, you can easily press Ctrl-C. This immediately quits the GP run. The Matlab command window will not be closed. If you do another run, it will be used again. This does not cause any troubles.

estimtf.m

An m-file which determines the transfer function between an input and output of a simulink model by applying white noise to the input channel. The transfer function is determined using the Matlab command TFE. See `estimtf.m` for required inputs and created outputs or look at a sample fitness-file `fitness.m` how it can be used.

simmodel.m

An m-file which can run multiple simulations of a Simulink model. All inputs of a model can be specified. See `simmodel.m` for all required inputs or look at a sample of a fitness-file how it can be used.

setp_3c.m

An m-file which generates a third order setpoint. The adjustable options are: amplitude, maximum velocity, maximum acceleration, maximum jerk, start time, end time and sample time. A time, position, velocity and acceleration signal is returned. See `setp_3c.m` for more info.

calcstop.dll

An S-function which is used in a simulink model. It terminates the simulation if a maximum running time is reached. The parameter of the simulink block is specified in seconds. This block should be present in every system model the fitness-function uses for simulations. Without this, complex models will take a lot of time to finish. The reason is often that the model contains stiff differential equations which are hard to solve. Such models have to be thrown away and are given a bad fitness.

protecteddiv.dll

An S-function which is used instead of a pure division block. When dividing by zero an error occurs. This is prevented by using this block. The denominator saturates if the absolute value is smaller than $1 \cdot 10^{-6}$ so it will not become 0 which prevents division by zero.

gp.ini

An ini-file to set parameters which are used by the GP program `gp4so.exe`. The file is divided into the sections: [GP Settings], [Genetic Operations], [Nodes], [Node Settings], [Matlab/Simulink Settings] and [GA optimization]. To understand and get a feeling how to use these settings, you have to explore the working principle of GP first. The sections and keys in the ini-file will be described next:

Section: [GP Settings]

Key	Description	Default value
Individuals	Size of the population	200
TournamentSelect	Amount of candidates used for Tournament Selection when selecting parents and when putting children back in the population	2
SelectionAmount	Amount of offspring per generation	4
MaxTreeDepth	Limit of amount of simulink-blocks after each other	6
TerminationFitness	Fitness value when the GP run stops	0
MaxGenerations	Generation when the GP run stops	500000

Section: [Genetic Operations]

Key	Description	Default value
Crossover	Exchanges a subtree from two parents	0.7
BranchMutation	Creates a new tree at a randomly chosen point	0.1
BranchAddition	Adds a tree by inserting a block with two inputs at the top and let a new branch grow at the free input	0.01
PointMutation	Exchanges a node by a new node	0.2
GA Optimization	Optimization of the parameters of an individual	0.1

The key-values indicate the chance that it is used. The chances are independent so it is possible that more genetic operations are applied to the same individual or even none. The GA optimization algorithm is also seen as a genetic operation, it only optimizes the values of an individual.

Section: [Nodes]

Key	Description	Default value
Add	Addition node	1.0
Subtract	Subtraction node	1.0
Multiply	Multiplication node	1.0
Divide	Division node (protecteddiv used, see Appendix A)	1.0
Integrate	Integration node	1.0
Differentiate	Differentiation node (including a pole to make it proper, see Section 4.4)	1.0
TF 1st order	Transfer Function 1st order	1.0
TF 2nd order	Transfer Function 2nd order	1.0
Gain	Gain node	1.0
Constant	Constant node	1.0
Time Delay	Time delay node	1.0
Input0, Input1 ... Input7	Input nodes	1.0

The key-values indicate the relative chance it is used (relative with respect to other node types). Setting it to zero it is not used. For a more detailed description of these node, see Section 4.4.

Section: [Node Settings]

Key	Description	Default value
AmountOfInputs	Amount of input-blocks available	1
MaxInitValue	Maximum initialization value of parameters in a new created node	0
OnlyNegativePoles	Set to true to enforce negative poles in the transfer functions and differentiator	false
OnlyNegativeZeros	Set to true to enforce negative zeros in the transfer functions	false

Be sure you added "AmountOfInputs" input-nodes to the section [Nodes], starting with "Input0", "Input1" ... "Input7". At the moment a maximum of 8 inputs is possible. The meaning of these inputs is determined by the fitness-function.

Section: [Matlab/Simulink Settings]

Key	Description	Default value
FitnessMFile	Name of the fitness-function	fitness
SimWorkDir	Work directory, location of fitness-function, dll's and other needed files	.
ModelSaveDir	Sub directory of the Work dir to save improved models during a GP run	mdls
BadSimPenalty	Fitness penalty for bad models	1e10
BlockSize	Size of a block in simulink	25
BlockSpace	Space between blocks in simulink	50

Section: [GA optimization]

Key	Description	Default value
Alpha	Increase/Decrease of the search standard deviation	1.5
Parents	Amount of parents	1
Recombi	Amount of recombinations	1
PopSize	Population size	10
MaxGen	Maximum amount of generations	10
InitStepSizeFactor	Initial standard deviation factor, the key-value is multiplied with the parameter	0.1
KeepBest	set to true to return the best individual of all generation set to false to return the best individuals of the last generation	false

fitness m-file

The fitness-function is an m-file which determines how good a model is. The input is a Simulink model with a fixed amount of inputs. The output is a number which indicates how good a model is, the fitness value. A small number is good and a large number is bad. 0 is the minimum and "BadSimPenalty" (10^{10}) is the maximum. This fitness value is used in the GP and GA algorithms for minimization. So it is crucial to define a good fitness-function. It should always return a valid number and your demands have to cover all possibilities in the search space.

```
fitness.m
plant_2nd_order.mdl
fitness_pato.m
pato_motor.mdl
pato_load.mdl
```

These are working example files to get a better insight how to set up such files. Before using them, check the `gp.ini` file if the correct fitness-function has been selected.

gp_model

The GP program creates a Simulink model every time it wants to determine a fitness of a particular individual. This model is created during runtime and will not be saved on the hard disk. Anyhow, it may not be saved, else a problem of equally named models occurs. So look out when checking intermediate results. Do not save the Simulink models when closing them.

```
temp_modelxxx.mdl
fitness.log
```

When a better model has been found it will be saved in the "mdls"-subdirectory. The model is saved as `temp_modelxxx.mdl` with `xxx` a successive number. Also its fitness and fitness evaluation count is appended to `fitness.log`. With this, the fitness progression can be examined later together with the structure of accompanying models when fitness improvements occur.

```
<savedir>\<savename>.mdl
<savedir>\<savename>_fitness.log
<savedir>\<savename>_fitness.m
<savedir>\<savename>_used_system.mdl
<savedir>\<savename>_logdata.txt
loghistory.txt
```

If you save the best model at the end of the GP program, the files above are saved. The model itself and the fitness progression file are copied from the "mdls"-subdirectory. The GP model is the temp_modelxxx.mdl with the highest number. Furthermore, the fitness-file and the system model are saved from the work-directory. Finally, some settings and results are saved into <savename>_logdata.txt and this is also appended to loghistory.txt. All these files are saved for two reasons: The result can be evaluated independently of the current state of the GP program. And secondly, all important settings are saved, so in a later stage a similar GP run can be performed.

D.1.3 Node types

Below, the Simulink blocks are presented with their characteristics. "uniform $[0,n]$ " means that the parameter is initialized by drawing a random number out of a uniform probability distribution between 0 and n (n is the "MaxInitValue" specified in gp.ini.). "loguniform $[\frac{1}{n},n]$ " means that a number is drawn between $\frac{1}{n}$ and n such that the chance that $c_1 > c_2$ is equal to $c_1 < \frac{1}{c_2}$ with c_1 and c_2 drawings from a loguniform probability distribution between $\frac{1}{n}$ and n .

Add:



$$y(t) = u_1(t) + u_2(t)$$

Subtract:



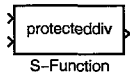
$$y(t) = u_1(t) - u_2(t)$$

Multiply:



$$y(t) = u_1(t)u_2(t)$$

Divide:



$$y(t) = \frac{u_1(t)}{u_2(t)}$$

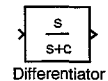
A protected division S-function is introduced because a normal division block would produce "division-by-zero"-problems in case of $u_2(t) = 0$.

Integrate:



$$y(s) = \frac{1}{s}u(s)$$

Differentiate:



$$y(s) = \frac{s}{s+c}u(s)$$

c : uniform $[0, n]$

This block, and also all TF-blocks use such an initialization so the largest initial pole is $-n$. It is wise to chose n somewhere below the sampling frequency (in rad/s). Note that this does not restrict creation of poles or zeros above a frequency n due to numerical optimization. The standard differentiation-block in Simulink ("du/dt") was not used because of numerical problems. The cause of this is that this block is not proper and will cause problems when solving the differential equation numerically. The gain goes to infinity with increasing frequency. Instead of this, a proper transfer-function with a pole is used. Below that pole-frequency it is equal to a pure differentiator.

Gain:

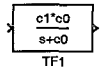


$$y(t) = cu(t)$$

$$c: \text{loguniform } \pm[\frac{1}{n}, n]$$

The parameter of a Gain-block is initialized using the loguniform probability distribution in order to keep the chance of a gain larger than 1 equal to the chance of a gain smaller than 1. This is done to avoid the assumption that a gain is mostly used for amplification.

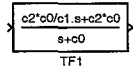
TF 1st order:



$$y(s) = c_1 \frac{1}{s/c_0 + 1} u(s)$$

$$c_0: \text{uniform } [0, n]$$

$$c_1: \text{loguniform } \pm[\frac{1}{n}, n]$$



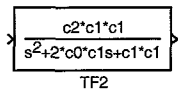
$$y(s) = c_2 \frac{s/c_1 + 1}{s/c_0 + 1} u(s)$$

$$c_0, c_1: \text{uniform } [0, n]$$

$$c_2: \text{loguniform } \pm[\frac{1}{n}, n]$$

The TF-blocks are scaled in such a way that the static gain is 1. An extra gain with a loguniform probability distribution is added to give the transfer function more freedom so it can have any shape possible for such a transfer function. Further, "TF 1st order" is one block with two possibilities. At the initialization of the block, one of the two possibilities will be chosen randomly. This is also done for "TF 2nd order" which has three possibilities. The reason of this construction is that only proper transfer functions can be made with Simulink. Without such a construction, it is not possible to place real and complex zeros and poles arbitrarily.

TF 2nd order:

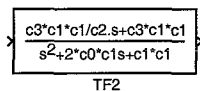


$$y(s) = c_2 \frac{c_1^2}{s^2 + 2c_0c_1s + c_1^2} u(s)$$

$$c_0: \text{uniform } [0, 1]$$

$$c_1: \text{uniform } [0, n]$$

$$c_2: \text{loguniform } \pm[\frac{1}{n}, n]$$



$$y(s) = c_3 c_1^2 \frac{s/c_2 + 1}{s^2 + 2c_0c_1s + c_1^2} u(s)$$

$$c_0: \text{uniform } [0, 1]$$

$$c_1, c_2: \text{uniform } [0, n]$$

$$c_3: \text{loguniform } \pm[\frac{1}{n}, n]$$

$$\frac{c_4^2 c_1^2 c_3 / c_3 s^2 + 2 c_4^2 c_1^2 c_3 c_2 s + c_4^2 c_1^2 c_3}{s^2 + 2 c_0 c_1 s + c_1^2}$$

TF2

$$y(s) = c_4 \frac{c_1^2 s^2 + 2 c_2 c_3 s + c_3^2}{c_3^2 s^2 + 2 c_0 c_1 s + c_1^2} u(s)$$

$c_0, c_2: \text{uniform } [0, 1]$
 $c_1, c_3: \text{uniform } [0, n]$
 $c_4: \text{loguniform } \pm[\frac{1}{n}, n]$

The second order transfer functions are scaled differently with a similarity to the normalized form: $s^2 + 2\xi\omega_n s + \omega_n^2$. The parameter at the place of ξ is initialized between 0 and 1 to produce under-critical-damped systems which are mostly used in the field of system identification and controller design.

Constant:



Constant

$$y(t) = c$$

$$c: \text{loguniform } \pm[\frac{1}{n}, n]$$

Time Delay:



Time Delay

$$y(t) = u(t - c)$$

$$c: \text{uniform } [0, 10/n]$$

Assuming n lies near the sampling frequency, then the maximum initial time delay of this block is 10 samples.

Inputx:



Inputx

$$y(t) = u_x(t)$$

D.1.4 Performing a GP run

Steps for a successful GP run are:

1. Check all settings of `gp.ini` and adjust them where necessary.
2. Adjust the fitness-function according to your desires. This part is the most important and most difficult to do, so spend enough time for this. You need to specify what you really want which sounds much easier than it actually is. One value has to be returned and it should be a good indication how good a model is. Furthermore, the fitness-function should always return a value, else undefined behavior can occur.
3. Run the `gp4so.exe` and wait! Better think in days instead of minutes. Every fitness evaluation, a simulink model has to be created and simulated so that just takes a lot of time. Intermediate results can be checked as described before.
4. If a termination criterion has been reached or you have pressed a key, you will be asked if you really want to abort the GP run. If you type "y", you can save the best model including additional data.
5. Finally, the saved result can be evaluated and validated.

D.1.5 Troubleshooting

- If the Matlab engine does not start, check if you have Matlab version 6.1 or 6.5. The application does not work with other versions at the moment.
- When you get a "License Manager Error" from Matlab you do not have a correct `license.dat` file in the `matlabdir/bin/win32`-directory. This problem occurs for example when this file is accessed via a network for multiple use. Contact the system administrator about this to copy the file to your local computer.
- If at the startup, some warnings occur about missing items in `gp.ini`, you should check if you spelled those names correctly.
- If strange fitness-numbers occur or the GP program throws an error or just quits immediately, probably the fitness-function was not setup correctly. The fitness-function should always return a number, which must be between 0 and 10^{10} ("BadSimPenalty") in every possible situation. For example, simulation errors which are realistic to occur because of the randomly created models can be caught by a try-catch construction.
- If the performance of the model is bad after running the GP program for a long time, check the elements of the fitness-function and the settings in `gp.ini`. The fitness-function shapes the search space and the settings of GP determine how to explore the search space. Both items are crucial for convergence to a good solution.

D.2 C++ code description

The C++-application `gp4so` (Genetic Programming For System Optimization) is based on the EO-library (Evolving Objects-Library for C++ version 0.9.2). This is an object oriented and template-based library which is developed for implementing Evolutionary Algorithms. It provides a lot of classes that can be of use. It is not a ready-made application, so making it dedicated for a specific task it needs to be extended with self-made classes. To get familiar with using the EO-library has proven to be quite difficult and will take some time.

D.2.1 C++-files

Besides of the EO-library files some additional files are needed to create the `gp4so-executable`:

- `gp4so.cpp`
- `engine.cpp`, `engine.h`
- `inireader.cpp`, `inireader.h`
- `logdata.cpp`, `logdata.h`
- `matlabcom.cpp`, `matlabcom.h`
- `node.cpp`, `node.h`
- `nodeproperties.cpp`, `nodeproperties.h`
- `settings.cpp`, `settings.h`
- `fitness.h`
- `eogaoperator.h`
- `eogencontinue2.h`
- `eomutationoperators.h`
- `eoparsetreedepthinit2.h`
- `eosgatransform2.h`

D.2.2 Description

`gp4so.cpp`

This is the main application. During exploration of the EO-library and the creation of the first application, a sample application was used which is dedicated to symbolic regression. This was quite a mess which has been partly reduced in the current application. It is still no optimal architecture or fully user-friendly programmed, but it has improved a lot. It just takes quite a time the EO-library begins to live. Besides, the EO-library is still in development and not finished.

`engine.cpp`, `engine.h`

This is the class which makes the Matlab engine available. It reads the Matlab-path from the windows registry and uses `libeng.dll` and `libmx.dll` from Matlab to connect to some necessary functions to operate the Matlab engine. At the moment it is only compatible with Matlab 6.1 and Matlab 6.5.

`inireader.cpp`, `inireader.h`

This class enables reading of the data types double, int, bool and string from an ini-file. The ini-file should consist of sections (with [...]) and keys belonging to a section. The value of a key is separated with "=". With these parameters, the program can be tuned without recompiling the program.

`logdata.cpp`, `logdata.h`

This class collects data about settings, results, run-info etc. and will be saved at the end of a GP run.

`matlabcom.cpp`, `matlabcom.h`

This class provides all the functions needed for communication between the GP program and Matlab/Simulink. It also converts a GP individual to a Simulink model.

`node.cpp`, `node.h`

This class represents a node in a tree (or individual). Such a node corresponds with a Simulink block. But it is not exactly one on one. For example, a tree can have more of the same input, but a Simulink model can only have one of each input (which will then split into more lines from one inport-block).

`nodeproperties.cpp`, `nodeproperties.h`

This class contains all characteristics of all node types. All nodes written in `gp.ini` should be defined in this class to be effective. Defined characteristics are: type, name in `gp.ini`, amount of inputs, if it is an input, usage chance, amount of parameters, initialization of the parameters and needed Matlab-code to generate the block in a Simulink model. If you want a new Simulink block, this class only needs to be changed and the `gp.ini` should contain the block.

`settings.cpp`, `settings.h`

This class is more like a collection of most settings retrieved from `gp.ini` and the `CNodeProperties`-class. At a lot of places the same settings are needed. By concentrating all these settings it is easily portable and you only have to define a setting once.

`fitness.h`

This template class determines the fitness of an individual. So it firstly creates a model, then it simulates that model and finally returns the fitness value. To do all this it makes use of the `CMatlabCom`-class.

`eogaoperator.h`

This template class implements the GA optimization algorithm. It is implemented as a genetic operation. The only difference is that it creates models and does fitness-evaluations. The other genetic operations do not do that. They only change a part of the individual and after all genetic operations are applied the new fitness of all offspring is calculated.

`eogencontinue2.h`

This template class is a changed file from the EO-library, because some more features are required. Its mainly about adding more termination criteria and that a confirmation will be asked if a termination criterion has been reached.

`eomutationoperators.h`

This header file contains some modified genetic operations: `eoPointMutation2`, `eoSubtreeXOver2`, `eoBranchMutation2` and a new genetic operation: `eoBranchAddition`. A reason that the first three operators are changed is that the same node type does not have to be identical because it can contain numerical values. Another reason is that there was an inconsistency in the EO-library. When initializing a tree its depth is limited (this has always been assumed and can be set in `gp.ini`). When the original genetic operation was applied the amount of nodes was limited instead of the depth.

`eoparsetreedepthinit2.h`

This template class randomly creates a new individual. It builds a tree by selecting available nodes using the usage chances. Furthermore, it initializes all existing parameters of an individual according to their probability distribution defined in `CNodeProperties`.

`eosgatransform2.h`

This template class applies all genetic operations to produce new offspring. This class has been changed from the original one, because it only could handle two genetic operations.

D.2.3 Known bugs/possible improvements/experience

- The class `FitnessType` is not used as it is meant to be. Its purpose is to define whether GP should minimize or maximize the fitness value. In the whole program, minimizing the fitness value has been assumed. The fitness is represented as a double data type and not as a more sophisticated class like `FitnessType`.
- The authors experience is that the EO-library is very big and looks like spaghetti for a non-OO-programmer the first month. So it is very difficult to use it as supposed to. A result is that some parts of the code are not very nicely coded because you have to stick to the way the EO-library is coded. The author was previously not used to program object oriented. At some places this is visible, like concentrating all settings in `CSettings` instead of localizing those settings in the places where they are actually needed.